LEF 5.8 C/C++ Programming Interface (Open Licensing Program)

Product Version 5.8
May 2017

Contents

Preface

This document describes the C and C++ programming interface used to read and write Cadence® Library Exchange Format (LEF) files. You should be an experienced C++ or C programmer and be familiar with LEF file structure to read this manual.

What's New

For information on what is new or changed in the LEF programming interface for version 5.8, see *What's New in LEF C/C++ Programming Interface.*

For information on what is new or changed in the DEF programming interface for version 5.8, see *What's New in DEF C/C++ Programming Interface*.

For information on what is new or changed in LEF and DEF for version 5.8, see *What's New in LEF/DEF*.

Related Documents

The LEF C/C++ programming interface lets you create programs that read and write LEF files. For more information about the Design Exchange Format (DEF) file syntax, see the *LEF/DEF Language Reference*.

Typographic and Syntax Conventions

This list describes the conventions used in this manual.

`text`

Words in `monospace` type indicate keywords that you must enter literally. These keywords represent language tokens.

`variable`

Words in *`italics`* indicate user-defined information for which you must substitute a name or a value.

`int`

Specifies an integer argument

`num`

Some LEF classes can be defined more than once. A statement that begins with the identifier *num* represents a specific number of calls to the particular class type.

`{ }`

Braces enclose each entire LEF class definition.

`|`

Vertical bars separate possible choices for a single argument. They take precedence over any other character.

`[ ]`

Brackets denote optional arguments. When used with vertical bars, they enclose a list of choices from which you can choose one.

4/12/17

1

Introduction

This chapter contains the following sections:

Overview

This manual describes the application programming interface (API) routines for the following Cadence® Library Exchange Format (LEF) components:

- LEF reader
- LEF writer

Cadence Design Systems, Inc. uses these routines internally with many tools that read and write LEF. The API supports LEF version 5.8, but also reads earlier versions of LEF.

You can use the API routines documented in this manual with tools that write these older versions, as long as none of the tools in an interdependent flow introduce newer constructs.

Note: The writer portion of the API does not always optimize the LEF output.

LEF Reader Working Modes

The LEF reader can work in two modes - compatibility mode and session-based mode.

- Compatibility mode (session-less mode) - This mode is compatible with the old parser behavior. You can call the parser initialization once with `lefrInit()`, adjust parsing settings and initialize the parser callbacks any time. The properties once defined in `PROPERTYDEFINITIONS` sections will be also defined in all subsequent file reads.

- Session-based mode - This mode introduces the concept of the parsing session. In this mode, the order of calling parsing configuration and processing API is strict:

  a. Parser initialization: Call `lefrInitSession()` instead of `lefrInit()` to start a new parsing session and close any old parsing session, if opened.

  b. Parser configuration: Call multiple callback setters and parsing parameters setting functions.

  c. Data processing: - Do one or multiple parsing of LEF files with the `lefrRead()` function.

  d. Cleaning of the parsing configuration: Call the `lefrClear()` function (optional). The call releases all parsing session data and closes the parsing session. If this is skipped, the data cleaning and the session closing is done by the next `lefrInitSession()` call.

  In the session-based mode, the properties once defined in `PROPERTYDEFINITIONS` remain active in all the LEF file parsing cycles in the session and the properties definition data is cleaned when the parsing session ends.

  The session-based mode does not require you to call callbacks and property unsetter functions. All callbacks and properties are set to default by the next `lefrInitSession()` call.

  The session-based mode allows you to avoid the lasting `PROPERTYDEFINITIONS` data effect when not required as you can just configure your application to parse one file per session.

By default, the LEF parser works in the compatibility mode. To activate the session-based mode, you must use `lefrInitSession()` instead of `lefrInit()`.

Note: Currently, the compatibility mode can be used in all old applications where the code has not been adjusted. The `lef2oa` translator has already been adjusted to use the session-based parsing mode.

Comparison Utility

The LEF file comparison utility, `lefdefdiff`, helps you verify that your usage of the API is consistent and complete. This utility reads two LEF files, generally an initial file and the resulting file from reading in an application, then writes out a LEF file. The comparison utility reads and writes the data so that the UNIX `diff` utility can be used to compare the files.

Because the LEF file comparison utility works incrementally (writing out as it operates), the size of files it can process has no limitations. However, large files can have performance restrictions. In general, this utility is intended only to verify the use of the API; that is, the utility is not a component of a production design flow.

Compressed LEF Files

The LEF reader can parse compressed LEF files. To do so, you must link the `liblef.a` and `liblefzlib.a` libraries.

A zlib compression library is also required in order to read compressed LEF files. The zlib source code is free software that can be downloaded from www.gnu.com.

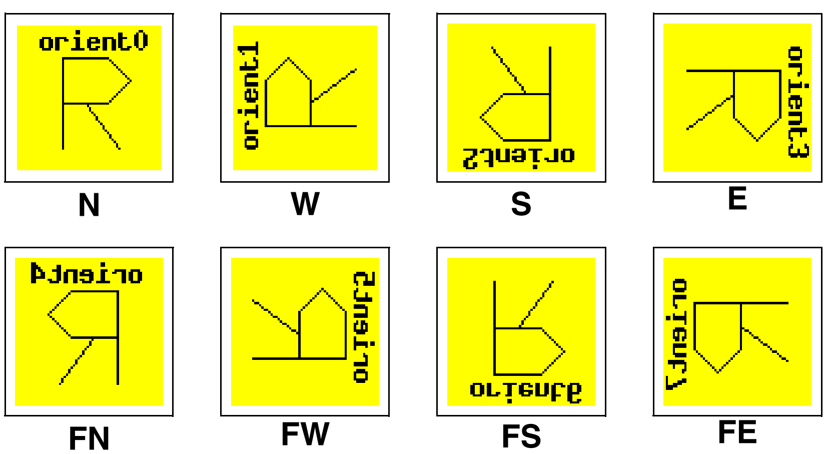
For information on compressed file routines, see "LEF Compressed File Routines."

Orientation Codes

Orientation codes are used throughout the LEF reader routines. The orientation codes are the same for all routines.

A number from 0 to 7, corresponding to the compass direction orientations, represents the orientation of a site or component. The following figure shows the combination of mirroring and rotation that is used for each of the eight possible orientations.

.



orient 0 = N

orient 1 = W

orient 4 = FN

orient 5 = FW

orient 2 = S	orient 6 = FS
orient 3 = E	orient 7 = FE

Note: The location given is the lower left corner of the resulting site or component after the mirroring and rotation are applied. It is *not* the location of the origin of the child cell.

2

LEF Reader Setup and Control Routines

The Cadence® Library Exchange Format (LEF) reader provides several routines to initialize the reader and set global variables that are used by the reader.

The following routines set options for reading a LEF file.

Calling the API Routines

Follow these steps to use the application programming interface (API) routines.

1. Call the `lefrInit()` routine. You must call this routine first.

2. Call the callback registration routines for those constructs your application uses.
3. Call any additional setup and control routines required to prepare for reading the LEF file.
4. Call the `lefrRead()` routine to start reading the LEF file.

As each construct in the LEF file is read, the reader calls the appropriate registered callbacks. These callbacks handle storing the associated data in a format appropriate for the application. The callbacks can call additional setup and control routines for the LEF reader as required.

For examples of API routine usage, see Appendix A, "LEF Reader and Writer Examples."

LEF API Routines

The following LEF reader setup and control routines are available in the API.

lefrInit

Initializes internal variables in the LEF reader. You must use this routine before using `lefrRead`. You can use routines to set callback functions before or after this routine.

Syntax

```
int lefrInit()
```

lefrInitSession

Starts a new parsing session and closes any old parsing session, if open. You must use this routine before using `lefrRead`.

Syntax

```
int lefrInitSession(
     int startSession = 1)
```

Arguments

startSession

Boolean. If is non-zero, performs the parser initialization in session-based mode; otherwise, the function will initialize the parsing in the compatibility mode, working exactly as `lefrInit()` call.

lefrClear

Releases all parsing session data and closes the parsing session. if the call to `lefrClear()` is skipped, the data cleaning and the session closing is done by the next `lefrInitSession()` call.

Syntax

```
int lefrClear()
```

lefrGetUserData

Retrieves the user-provided data. The LEF reader returns an opaque `lefiUserData` pointer, which you set using `lefrSetUserData`. You can set or change the user data at any time with the `lefrSetUserData` and `lefrGetUserData` calls. Every callback returns the user data as the third argument.

Syntax

```
lefiUserData lefrGetUserData()
```

lefrPrintUnusedCallbacks

Prints all callback routines that are not set but have constructs in the LEF file.

Syntax

```
void lefrPrintUnusedCallbacks(FILE* f)
```

lefrRead

Specifies the LEF file to read. If the file parses with no errors (that is, all callbacks return condition codes that indicate success), this routine returns a value of `0`.

Syntax

```
int lefrRead(
     FILE* file,
     const char* fileName,
     lefiUserData* data)
```

Arguments

file

Specifies a pointer to an already open file. This allows the parser to work with either a disk file or a piped stream. This argument is required. Any callbacks that have been set will be called from within this routine.

fileName

Specifies a UNIX filename using either a complete or a relative path specification.

data

Specifies the data type. For information about the `lefiUserData` type, see "lefiUserData" on page 89.

lefrRegisterLef58Type

Registers new LEF layers LEF58_TYPE – TYPE pairs. As LEF syntax requires that any layer LEF58_TYPE can be used only for certain layer types, you have to set a number of allowed layer LEF58_TYPE – TYPE pairs, calling the function several times (if necessary). For example, to register a new LEF58_TYPE `XXX` for the `CUT` and `ROUTING` type layers, you have to call the API twice:

```
lefrRegisterLef58Type('XXX', 'CUT');

lefrRegisterLef58Type('XXX', 'ROUTING');
```

Use this feature only for the development of new 'experimental' types, which can now be introduced without parser code update. All types mentioned in LEF documentation are already pre-set and do not require to be registered.

Syntax

```
void lefrRegisterLef58Type(
     const char* lef58Type,
     const char* layerType);
```

Arguments

lef58Type

Specifies the LEF layer lef58Type.

layerType

Specifies the LEF layer type.

lefrReset

Resets all of the internal variables of the LEF reader to their initial values.

Syntax

```
int lefrReset(void)
```

lefrSetCommentChar

Changes the character used to indicate comments in the LEF file.

Syntax

```
void lefrSetCommentChar(char c)
```

c

Specifies the comment character. The default character is a pound sign (#).

lefrSetRegisterUnusedCallbacks

Keeps track of all the callback routines that are not set. You can use this routine to keep track of LEF constructs that are in the input file but do not trigger a callback. This statement does not require any additional arguments.

Syntax

```
void lefrSetRegisterUnusedCallbacks(void)
```

lefrSetShiftCase

Allows the parser to upshift all names if the LEF file is case insensitive.

Syntax

```
void lefrSetShiftCase(void)
```

lefrSetUserData

Sets the user-provided data. The LEF reader does not look at this data, but passes an opaque `lefiUserData` pointer back to the application with each callback. You can set or change the user data at any time using the `lefrSetUserData` and `lefrGetUserData` routines. Every callback returns the user data as the third argument.

Syntax

```
void lefrSetUserData(
     lefiUserData* data)
```

Arguments

data
Specifies the user-provided data.

lefrSetVersionValue

Sets a default version number for a LEF file that does not conrtain a `VERSION` statement.

Syntax

```
void lefrSetVersionValue(
     char* version)
```

Arguments

version

Specifies the version number to assign to the LEF file.

Examples

The following example shows how to initialize the reader.

```
int setupRoutine() {
         FILE* f;
         int   res;
         int   userData = 0x01020304;
         ...

         // Initialize the reader.  This routine is called first.
         lefrInit();

         // Set user data
          lefrSetUserData ((void*)3);

         // Open the lef file for the reader to read
         if ((f = fopen("lefInputFileName","r")) == 0) {
             printf("Couldn't open input file '%s'\n",
               "lefInputFileName");
             return(2);
         }

         // Invoke the parser
         res = lefrRead(f, "lefInputFileName", (void*)userData);
         if (res != 0) {
             printf("LEF parser returns an error\n");
             return(2);
         }

     fclose(f);

     return 0;}
```

3

LEF Reader Callback Routines

The Cadence® Library Exchange Format (LEF) reader calls all callback routines when it reads in the appropriate part of the LEF file. Some routines, such as the version callback, are called only once. Other routines can be called more than once.

This chapter contains the following sections:

Callback Function Format

All callback functions have the following format:

```
int UserCallbackFunction(
     lefrCallbackType_e callbackType
     data_type* LEF_data
     lefiUserData data)
```

Each user-supplied callback routine is passed three arguments.

Callback Type

The `callBackType` argument is a list of objects that contains a unique number assignment for each callback from the parser.This list allows you to use the same callback routine for different types of LEF data. For examples, see Appendix A, “LEF Reader and Writer Examples.”

LEF_Data

The *LEF_data* argument provides the data specified by the callback. Data types returned by the callbacks vary for each callback. Examples of the types of arguments passed include `const char*`, `double`, `int`, and `defiProp`. Two points to note:

- The data returned in the callback is not checked for validity.
- If you want to keep the data, you must make a copy of it.

User Data

The *data* argument is a four-byte data item that is set by the user. The LEF reader contains only user data. The user data is most often set to a pointer to the library data so that it can be passed to the routines. This is more effective than using a global variable.

The callback functions can be set or reset at any time. If you want a callback to be available when the LEF file parsing begins, you must set the callback before you call `lefrRead`.

Note: You can unset a callback by using the set function with a null argument.

Callback Types and Setting Routines

You must set a callback before you can use it. When you set a callback, the callback routine used for each type of LEF information is passed in the appropriate setting routine. Each callback routine returns a callback type.

The following table lists the LEF reader callback setting routines and the associated callback types. The contents of the setting routines are described in detail in the section "User Callback Routines" on page 36.

LEF Information	Setting Routine	Callback Type
Bus Bit Characters	`void lefrSetBusBitCharsCbk (lefrStringCbkFnType);`	`lefrBusBitCharsCbkType`
Clearance Measure	`void lefrSetClearanceMeasureCbk (lefrStringCbkFnType);`	`lefrClearanceMeasureCbkType`
Density	`void lefrSetDensityCbk (lefrDensityCbkFnType)`	`lefrDensityCbkType`

LEF Information	Setting Routine	Callback Type
Divider Character	`void lefrSetDividerCharCbk (lefrStringCbkFnType);`	`lefrDividerCharCbkType`
Extensions	`void lefrSetExtensionCbk (lefrStringCbkFnType)`	`lefrExtensionCbkType`
FixedMask	`void lefrFixedMaskCbk (lefrIntergerCbkFnType)`	`lefrFixedMaskCbkType`
Library End Statement	`void lefrSetLibraryEndCbk (lefrVoidCbkFnType);`	`lefrLibraryEndCbkType`
Layer	`void lefrSetLayerCbk (lefrLayerCbkFnType);`	`lefrLayerCbkType`
Macro Beginning	`void lefrSetMacroBeginCbk (lefrStringCbkFnType);`	`lefrMacroBeginCbkType`
Macro	`void lefrSetMacroCbk (lefrMacroCbkFnType);`	`lefrMacroCbkType`
Macro Class Type	`void lefrSetMacroClassTypeCbk (lefrStringCbkFnType);`	`lefrMacroClassTypeCbkType`
Macro End	`void lefrSetMacroEnd (lefrStringCbkFnType);`	`lefrMacroEndCbkType`
Macro Fixed Mask	`void lefrMacroFixedMaskCbk (lefrIntergerCbkFnType)`	`lefrMacroFixedMaskCbkType`
Macro Foreign	`void lefrSetMacroForeignCbk (lefrMacroForeignCbkFnType);` `void lefrUnsetMacroForeignCbk();`	`lefrMacroForeignCbkFnType`
Macro Origin	`void lefrSetMacroOriginCbk (lefrMacroNumCbkFnType);`	`lefrMacroOriginCbkType`
Macro Obstruction	`void lefrSetObstructionCbk (lefrObstructionCbkFnType);`	`lefrObstructionCbkType`
Macro Pin	`void lefrSetPinCbk (lefrPinCbkFnType);`	`lefrPinCbkType`

LEF Information	Setting Routine	Callback Type
Macro Site	`void lefrSetMacroSiteCbk (lefrMacroSiteCbkFnType);`	`lefrMacroSiteCbkFnType`
	`void lefrUnsetMacroSiteCbk();`	
Macro Size	`void lefrSetMacroSizeCbk (lefrMacroNumCbkFnType);`	`lefrMacroSizeCbkType`
Manufacturing Grid	`void lefrSetManufacturingCbk (lefrDoubleCbkFnType);`	`lefrManufacturingCbkType`
Maximum Via Stack	`void lefrSetMaxStackViaCbk (lefrMaxStackViaCbkFnType);`	`lefrMaxStackViaCbkType`
Nondefault Rules	`void lefrSetNonDefaultCbk (lefrNonDefaultCbkFnType);`	`lefrNonDefaultCbkType`
Property Definitions Beginning	`void lefrSetPropBeginCbk (lefrVoidCbkFnType);`	`lefrPropBeginCbkType`
Property Definitions	`void lefrSetPropCbk (lefrPropCbkFnType);`	`lefrPropCbkType`
Property Definitions End	`void lefrSetPropEndCbk (lefrVoidCbkFnType);`	`lefrPropEndCbkType`
Same-Net Spacing Beginning	`void lefrSetSpacingBeginCbk (lefrVoidCbkFnType);`	`lefrSpacingBeginCbkType`
Same-Net Spacing	`void lefrSetSpacingCbk (lefrSpacingCbkFnType);`	`lefrSpacingCbkType`
Same-Net Spacing End	`void lefrSetSpacingEndCbk (lefrVoidCbkFnType);`	`lefrSpacingEndCbkType`
Site	`void lefrSetSiteCbk (lefrSiteCbkFnType);`	`lefrSiteCbkType`
Units	`void lefrSetUnitsCbk (lefrUnitsCbkFnType);`	`lefrUnitsCbkType`
Use Min Spacing	`void lefrSetUseMinSpacingCbk (lefrUseMinSpacingCbkFnType);`	`lefrUseMinSpacingCbkType`

LEF Information	Setting Routine	Callback Type
Version	`void lefrSetVersionCbk (lefrDoubleCbkFnType);`	`lefrVersionCbkType`
Version String	`void lefrSetVersionStrCbk (lefrStringCbkFnType);`	`lefrVersionStrCbkType`
Via	`void lefrSetViaCbk (lefrViaCbkFnType);`	`lefrViaCbkType`
Via Rule	`void lefrSetViaRuleCbk (lefrViaRuleCbkFnType);`	`lefrViaRuleCbkType`
Unused	`void lefrSetUnusedCallbacks (lefrVoidCbkFnType func);`	`lefrUnspecifiedCbkType`

Examples

The following example shows how to create a setup routine so the reader can parse the LEF file and call the callback routines you defined.

```
int setupRoutine() {
         FILE* f;
         int   res;
         int   userData = 0x01020304;
         ...

         // Initialize the reader.  This routine is called first.
         lefrInit();

         // Set the user callback routines
         lefrSetArrayBeginCbk(arrayBeginCB);
         lefrSetArrayCbk(arrayCB);
         lefrSetArrayEndCbk(arrayEndCB);
         lefrSetBusBitCharsCbk(busBitCharsCB);
         lefrSetCaseSensitiveCbk(caseSensCB);
         lefrSetDielectricCbk(dielectricCB);
         ...

         // Open the lef file for the reader to read
         if ((f = fopen("lefInputFileName","r")) == 0) {
             printf("Couldn't open input file '%s'\n",
               "lefInputFileName");
             return(2);
         }
```

```
// Invoke the parser
     res = lefrRead(f, "lefInputFileName", (void*)userData);
     if (res != 0) {
         printf("LEF parser returns an error\n");
         return(2);
     }

fclose(f);

return 0;}lefrUseMinSpacingCbkFnType
```

User Callback Routines

This section describes the following user callback routines:

- lefrViaCbkFnType on page 51
- lefrViaRuleCbkFnType on page 52
- lefrVoidCbkFnType on page 52

lefrDensityCbkFnType

Retrieves data from the `DENSITY` object from within the `MACRO` object. Use the arguments defined in the `lefiDensity` class to retrieve the data.

For syntax information about the LEF `MACRO` statement, see "Macro" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrDensityCbkFnType(
     lefrCallbackType_e typ
     lefiDensity* density
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrDensityCbkType` type. This allows you to verify within your program that this is a correct callback.

lefiDensity

Returns a pointer to a `lefiDensity` structure. For more information, see "lefiDensity" on page 68.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrDoubleCbkFnType

Retrieves different kinds of LEF data. The format of the data returned is always the same, but the actual data represented varies depending on the calling routine.

For more information about LEF syntax, see the *LEF/DEF Language Reference*.

Syntax

```
int lefrDoubleCbkFnType(
     lefrCallbackType_e typ,
     double number,
     lefiUserData* data)
```

Arguments

typ

Returns a type that varies depending on the callback routine used. The following types can be returned.

LEF Data	Type Returned
Manufacturing Grid	`lefrManufacturingCbkType`
Version	`lefrVersionCbkType`

number

Returns data that varies depending on the callback used. The following kinds of data can be returned.

LEF Data	Returns the Value of
Manufacturing Grid	*value* in the `MANUFACTURINGGRID` statement
Version	*number* in the `VERSION` statement

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrIntergerCbkFnType

Retrieves LEF data pertaining to fixed masks. The format of the data returned is always the same, but the actual data represented varies depending on the calling routine.

For more information about the `FIXEDMASK` statement, see "FIXEDMASK" in the ***LEF/DEF Language Reference***.

Syntax

```
int lefrIntergerCbkFnType
     leftCallbackType_e type,
     int number,
     lefiUserData* data)
```

Arguments

type

Returns a type that varies depending on the callback routine used. The following types can be returned.

LEF Data	Type Returned
FixedMask	`lefrFixedMaskCbkType`
Macro FixedMask	`lefrMacroFixedMaskCbkType`

number

Returns a type that varies depending on the callback used. The following kind of data can be returned.

Fixed mask: Does not allow mask shifting. All the LEF MACRO PIN MASK assignments must be kept fixed and cannot be shifted to a different mask, (1indicates not allowed, and 0 allowed).

Macro FixedMask: Indicates that the specified macro does not allow mask shifting. All the LEF PIN MASK assignments must be kept fixed and cannot be shifted to a different mask. (1 indicates not allowed, and 0 allowed).

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrLayerCbkFnType

Retrieves data from the `LAYER` object of the LEF file. Use the arguments defined in the `lefiLayer` class to retrieve the data.

For syntax information about the LEF `LAYER` statement, see "Layer (Cut)," "Layer (Masterslice or Overlap)," "Layer (Routing)," or "Layer (Implant)" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrLayerCbkFnType(
     lefrCallbackType_e typ,
     lefiLayer* layer,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrLayerCbkType` type. This allows you to verify within your program that this is a correct callback.

layer

Returns a pointer to a `lefiLayer` structure. For more information, see "lefiLayer" on page 61.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrMacroCbkFnType

Retrieves data from the `MACRO` object in the LEF file. Use the arguments defined in the `lefiMacro` class to retrieve the data.

For syntax information about the LEF `MACRO` statement, see "Macro" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrMacroCbkFnType(
     lefrCallbackType_e typ,
     lefiMacro* macro,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrMacroCbkType` type. This allows you to verify within your program that this is a correct callback.

macro

Returns a pointer to a `lefiMacro` structure. For more information, see “lefiMacro” on page 69.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrMacroForeignCbkFnType

Retrieves data for in-place processing of a `MACRO FOREIGN` statement. Use the arguments defined in the `lefiMacroForeign` class to retrieve the data.

For syntax information about the LEF `MACRO FOREIGN` statement, see "Macro" in the *LEF/ DEF Language Reference*.

Syntax

```
int lefrMacroForeignCbkFnType(
     lefrCallbackType_e typ,
     lefiMacroForeign* foreign,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrMacroForeignCbkType` type. This allows you to verify within your program that this is a correct callback.

foreign

Returns a pointer to a `lefiMacroForeign` structure. For more information, see lefiMacroForeign on page 70.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrMacroNumCbkFnType

Retrieves different kinds of Macro LEF data. The format of the data returned is always the same, but the actual data represented varies depending on the calling routine.

For syntax information about the LEF `MACRO` statement, see "Macro" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrMacroNumCbkFnType(
     lefrCallbackType_e typ,
     lefiNum num,
     lefiUserData* data)
```

Arguments

typ

Returns a type that varies depending on the callback routine used. The following types can be returned.

LEF Data	Type Returned
Macro Origin	`lefrMacroOriginCbkType`
Macro Size	`lefrMacroSizeCbkType`

num

Returns data that varies depending on the callback used. The following kinds of data can be returned.

LEF Data	Returns the Value of
Macro Origin	*value* for `ORIGIN` in the `MACRO` statement.
Macro Size	*value* for `SIZE` in the `MACRO` statement.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrMacroSiteCbkFnType

Retrieves data for in-place processing of a `MACRO SITE` statement. Use the arguments defined in the `lefiMacroSite` class to retrieve the data.

For syntax information about the LEF `MACRO FOREIGN` statement, see "Macro" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrMacroSiteCbkFnType(
     lefrCallbackType_e typ,
     lefiMacroSite* site,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrMacroSiteCbkType` type. This allows you to verify within your program that this is a correct callback.

site

Returns a pointer to a `lefiMacroSite` structure. For more information, see lefiMacroSite on page 71.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrMaxStackViaCbkFnType

Retrieves data from the `MAXVIASTACK` object in the LEF file. Use the arguments defined in the `lefiMaxStackVia` class to retrieve the data.

For syntax information about the LEF `NONDEFAULTRULE` statement, see "Maximum Via Stack" in the *LEF/DEF Language Reference*.

Syntax

```
lefrMaxStackViaCbkFnType(
     lefrCallbackType_e typ,
     lefiMaxStackVia* maxStack,
     lefiUserData data)
```

Arguments

typ

Returns the `lefrMaxStackViaCbkType` type. This allows you to verify within your program that this is a correct callback.

maxStack

Returns a pointer to a `lefiMaxStackVia` structure. For more information, see "lefiMaxStackVia" on page 88.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrNonDefaultCbkFnType

Retrieves data from the `NONDEFAULTRULE` object in the LEF file. Use the arguments defined in the `lefiNonDefault` class to retrieve the data.

For syntax information about the LEF `NONDEFAULTRULE` statement, see "Nondefault Rule" in the *LEF/DEF Language Reference*.

Syntax

```
lefrNonDefaultCbkFnType(
     lefrCallbackType_e typ,
     lefiNonDefault* def,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrNonDefaultCbkType` type. This allows you to verify within your program that this is a correct callback.

def

Returns a pointer to a `lefiNonDefault` structure. For more information, see "lefiNonDefault" on page 89.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrObstructionCbkFnType

Retrieves data from the `OBS` (macro obstruction) object within the `MACRO` object in the LEF file. Use the arguments defined in the `lefiObstruction` class to retrieve the data.

For syntax information about the LEF `OBS` statement, see "Macro Obstruction Statement" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrObstructionCbkFnType(
     lefrCallbackType_e typ,
     lefiObstruction* obs,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrObstructionCbkType` type. This allows you to verify within your program that this is a correct callback.

obs

Returns a pointer to a `lefiObstruction` structure. For more information, see "lefiObstruction" on page 74.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrPinCbkFnType

Retrieves data from the `PIN` object within the `MACRO` object in the LEF file. Use the arguments defined in the `lefiPin` class to retrieve the data.

For syntax information about the LEF `PIN` statement, see "Macro Pin Statement" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrPinCbkFnType(
     lefrCallbackType_e typ,
     lefiPin* pin,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrPinCbkType` type. This allows you to verify within your program that this is a correct callback.

pin

Returns a pointer to a `lefiPin` structure. For more information, see “lefiPin” on page 77.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrPropCbkFnType

Retrieves data from the `PROPERTYDEFINITIONS` object in the LEF file. Use the arguments defined in the `lefiProp` class to retrieve the data.

For syntax information about the LEF `PROPERTYDEFINITIONS` statement, see "Property Definitions" in the *LEF/DEF Language Reference*.

Syntax

```
lefrPropCbkFnType(
     lefrCallbackType_e typ,
     lefiProp* prop,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrPropCbkType` type. This allows you to verify within your program that this is a correct callback.

prop

Returns a pointer to a `lefiProp` structure. For more information, see "lefiProp" on page 91.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrSiteCbkFnType

Retrieves data from the `SITE` object in the LEF file. Use the arguments defined in the `lefiSite` class to retrieve the data.

For syntax information about the LEF `SITE` statement, see "Site" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrSiteCbkFnType(
     lefrCallbackType_e typ,
     lefiSite* site,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrSiteCbkType` type. This allows you to verify within your program that this is a correct callback.

site

Returns a pointer to a `lefiSite` structure. For more information, see "lefiSite" on page 95.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrSpacingCbkFnType

Retrieves data from the `SPACING` object of the LEF file. Use the arguments defined in the `lefiSpacing` class to retrieve the data.

For syntax information about the LEF `SPACING` statement, see "Samenet Spacing" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrSpacingCbkFnType(
     lefrCallbackType_e typ,
     lefiSpacing* spacing,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrSpacingCbkType` type. This allows you to verify within your program that this is a correct callback.

spacing

Returns a pointer to a `lefiSpacing` structure. For more information, see "lefiSpacing" on page 94.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrStringCbkFnType

Retrieves different kinds of LEF data. The format of the data returned is always the same, but the actual data represented varies depending on the calling routine.

For more information about LEF syntax, see the *LEF/DEF Language Reference*.

Syntax

```
int lefrStringCbkFnType(
     lefrCallbackType_e typ,
     const char* string,
     lefiUserData* data)
```

Arguments

typ

Returns a type that varies depending on the callback routine used. The following types can be returned.

LEF Data	Type Returned
Bus Bit Characters	`lefrBusBitCharsCbkType`
Clearance Measure	`lefrClearanceMeasureCbkType`
Divider Character	`lefrDividerCharCbkType`
Extensions	`lefrExtensionCbkType`
Macro Beginning	`lefrMacroBeginCbkType`
Macro Class Type	`lefrMacroClassTypeCbkType`
Macro End	`lefrMacroEndCbkType`
Version String	`lefrVersionStrCbkType`

string

Returns data that varies depending on the callback used. The following kinds of data can be returned.

LEF Data	Returns the value of
Bus Bit Characters	*delimterPair* in the `BUSBITCHARS` statement
Clearance Measure	Returns the string set for a `CLEARANCEMEASURE` statement
Divider Character	*character* in a `DIVIDERCHAR` statement
Extensions	Retruns the string set for an `EXTENSION` statement
Macro Beginning	*macroName* in a `MACRO` statement
Macro Class Type	Returns the string set for a `CLASS` statement in a `MACRO` statement
Macro End	`END` *macroName* in a MACRO statement
Version String	Returns the string set for a `VERSION` statement

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrUnitsCbkFnType

Retrieves data from the `UNITS` object in the LEF file. Use the arguments defined in the `lefiUnits` class to retrieve the data.

For syntax information about the LEF `UNITS` statement, see "Units" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrUnitsCbkFnType(
     lefrCallbackType_e typ,
     lefiUnits* units,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrUnitsCbkType` type. This allows you to verify within your program that this is a correct callback.

units

Returns a pointer to a `lefiUnits` structure. For more information, see "lefiUnits" on page 97.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrUseMinSpacingCbkFnType

Retrieves data from the `USEMINSPACING` object in the LEF file. Use the arguments defined in the `lefiUseMinSpacing` class to retrieve data.

For information about the LEF `USEMINSPACING` statement, see "Use Min Spacing" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrUseMinSpacingCbkFnType(
     lefrCallbackType_e typ,
     lefiUseMinSpacing* spacing,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrUseMinSpacingCbkFnType` type. This allows you to verify within your program that this is a correct callback.

spacing

Returns a pointer to a lefiUseMinSpacing structure. For more information, see "lefiUseMinSpacing" on page 99

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrViaCbkFnType

Retrieves data from the `VIA` object in the LEF file. Use the arguments defined in the `lefiVia` class to retrieve the data.

For syntax information about the LEF `VIA` statement, see "Via" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrViaCbkFnType(
     lefrCallbackType_e typ,
     lefiVia* via,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrViaCbkType` type. This allows you to verify within your program that this is a correct callback.

via

Returns a pointer to a `lefiVia` structure. For more information, see "lefiVia" on page 99.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrViaRuleCbkFnType

Retrieves data from the `VIARULE` object in the LEF file. Use the arguments defined in the `lefiViaRule` class to retrieve the data.

For syntax information about the LEF `VIARULE` statement, see "Via Rule" in the *LEF/DEF Language Reference*.

Syntax

```
int lefrViaRuleCbkFnType(
     lefrCallbackType_e typ,
     lefiViaRule* viaRule,
     lefiUserData* data)
```

Arguments

typ

Returns the `lefrViaRuleCbkType` type. This allows you to verify within your program that this is a correct callback.

viaRule

Returns a pointer to a `lefiViaRule` structure. For more information, see "lefiViaRule" on page 103.

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

lefrVoidCbkFnType

Marks the beginning and end of LEF objects. The format of the data returned is always the same, but the actual data represented varies depending on the calling routine.

For more information about LEF syntax, see the *LEF/DEF Language Reference*.

Syntax

```
int lefrVoidCbkFnType(
     lefrCallbackType_e typ,
     void* ptr,
     lefiUserData* data)
```

Arguments

typ

Returns a type that varies depending on the callback routine used. The following types can be returned.

LEF Data	Type Returned
Library End	`lefrLibraryEndCbkType`
Property Begin	`lefrPropBeginCbkType`
Property End	`lefrPropEndCbkType`
Spacing Begin	`lefrSpacingBeginCbkType`
Spacing End	`lefrSpacingEndCbkType`
Unused	`lefrUnspecifiedCbkType`

ptr

Returns nothing. (This is a placeholder value to meet the required three arguments for each routine).

data

Returns four bytes of user-defined data. User data is set most often to a pointer to the design data.

Examples

The following example shows a callback routine using `lefrCallbackType_e`, `char*`, and `lefiUserData`.

```
int macroBeginCB (lefrCallbackType_e type,
                       const char *macroName,
                       lefiUserData userData) {
```

```
    // Incorrect type was passed in, expecting the type lefiMacroBeginCbkType
    if (type != lefiMacroBeginCbkType) {
            printf("Type is not lefiMacroBeginCbkType,
             terminate parsing.\n");
            return 1;
        }


    // Expect a non null char* macroName
        if (!macroName || !*macroName) {
            printf("Macro name is null, terminate parsing.\n");
            return 1;
        }


    // Write out the macro name
        printf("Macro name is %s\n", macroName);
        return 0;}
```

The following callback routine has arguments of `lefrCallbackType_e`, `void*`, and `lefiUserData`.

```
int irdropEndCB (lefrCallbackType_e type,
                      void* ptr,
                      lefiUserData userData) {
        // Check if the type is correct
        if (type != lefrIRDropEndCbkType) {
            printf("Type is not lefrIRDropEndCbkType, terminate
             parsing.\n");
            return 1;
        }


    printf("IRDROP END\n");
        return 0;}
```

4

LEF Reader Classes

This chapter contains the following sections:

Introduction

Every statement in the Cadence® Library Exchange Format (LEF) file is associated with a LEF reader class. When the LEF reader uses a callback, it passes a pointer to the appropriate class. You can use the member functions in each class to retrieve data defined in the LEF file.

For a list of the LEF Reader Classes that correspond to LEF file syntax, see "LEF Reader Classes" on page 58.

Callback Style Interface

This programming interface uses a callback style interface. You register for the constructs that interest you, and the readers call your callback functions when one of those constructs is read. If you are not interested in a given set of information, you simply do not register the callback; the reader scans the information quickly and proceeds.

Important

Returned data is not static. If you want to keep the data, you must copy it.

Retrieving Repeating LEF Data

Many LEF objects contain repeating objects or specifications. The classes that correspond to these LEF objects contain an index and array of elements that let you retrieve the data iteratively.

You can use a `for` loop from `0` to the number of items specified in the index. In the loop, retrieve the data from the subsequent arrays. For example:

```
for (i = 0; i < layer->lefiLayer::numMinstep(); i++) {
       fprintf(fout, "  MINSTEP %g ", layer->lefiLayer::minstep(i));
       if (layer->lefiLayer::hasMinstepType(i))
          fprintf(fout, "%s ", layer->lefiLayer::minstepType(i));
       if (layer->lefiLayer::hasMinstepLengthsum(i))
          fprintf(fout, "LENGTHSUM %g ",
                  layer->lefiLayer::minstepLengthsum(i));
       fprintf(fout, ";\n");
    }
```

Deriving C Syntax from C++ Syntax

The Cadence application programming interface (API) provides both C and C++ interfaces. The C API is generated from the C++ source, so there is no functional difference. The C API has been created in a pseudo object-oriented style. Examining a simple case should enable you to understand the API organization.

The following examples show the same statements in C and C++ syntax.

C++ Syntax

```
class lefiSite {
       const char* name() const;
       int hasClass() const;
       const char* siteClass() const;
       double sizeX() const;
       double sizeY() const;
       int numSites() const;
       char* siteName(int index) const;
       int   siteOrient(int index) const;
       char* siteOrientStr(int index) const;

};
```

C Syntax

```
const char * lefiSite_name
       ( const lefiSite * this );

int lefiSite_hasClass
       ( const lefiSite * this );

const char * lefiSite_siteClass
       ( const lefiSite * this );

double lefiSite_sizeX
       ( const lefiSite * this );

double lefiSite_sizeY
       ( const lefiSite * this );

int lefiSite_numSites
       ( const lefiSite * this );

char * lefiSite_siteName
       ( const lefiSite * this, int index );

int lefiSite_siteOrient
       ( const lefiSite * this, int index );

char * lefiSite_siteOrientStr
       ( const lefiSite * this, int index );
```

The C routine prototypes for the API functions can be found in the following files:

lefiArray.h	lefiNonDefault.h	lefiViaRule.h
lefiCrossTalk.h	lefrCallBacks.h	lefiProp.h
lefrReader.h	lefiDebug.h	lefiDefs.h
lefwWriter.h	lefiKRDefs.h	lefiLayer.h
lefiUnits.h	lefiUser.h	lefiMacro.h
lefiUtil.h	lefiMisc.h	lefiVia.h

LEF Reader Classes

The following table lists the classes routines that apply to the LEF information.

LEF Information	**LEF Class**
Layer Classes	lefiAntennaModel lefiAntennaPWL lefiInfluence lefiLayer lefiLayerDensity lefiOrthogonal lefiParallel lefiSpacingTable lefiTwoWidths
Macro Data Classes	lefiDensity lefiMacro lefiMacroForeign lefiMacroSite lefiPoints
Macro Obstruction Class	lefiObstruction
Macro Pin Classes	lefiGeometries lefiPin lefiPinAntennaModel
Maximum Via Stack Class	lefiMaxStackVia
Miscellaneous Class	lefiUserData
Nondefault Rule Class	lefiNonDefault
Property Definition Classes	lefiProp lefiPropType
Same-Net Spacing Class	lefiSpacing
Site Classes	lefiSite lefiSitePattern
Units Class	lefiUnits
Use Min Spacing Class	lefiUseMinSpacing
Via Classes	lefiVia lefiViaLayer

LEF Information	LEF Class
Via Rule Classes	lefiViaRule lefiViaRuleLayer

Layer Classes

The LEF `LAYER` routines include the following classes:

- lefiAntennaModel on page 59
- lefiAntennaPWL on page 60
- lefiInfluence on page 61
- lefiLayer on page 61
- lefiLayerDensity on page 66
- lefiOrthogonal on page 66
- lefiParallel on page 67
- lefiSpacingTable on page 67
- lefiTwoWidths on page 68

lefiAntennaModel

Retrieves antenna model information from a `LAYER` section of the LEF file.

For syntax information about the LEF `LAYER` sections, see "Layer (Cut)," and "Layer (Routing)" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiAntennaModel {
     int hasAntennaAreaRatio() const;
     int hasAntennaDiffAreaRatio() const;
     int hasAntennaDiffAreaRatioPWL() const;
     int hasAntennaCumAreaRatio() const;
     int hasAntennaCumDiffAreaRatio() const
     int hasAntennaCumDiffAreaRatioPWL() const;
     int hasAntennaAreaFactor() const;
     int hasAntennaAreaFactorDUO() const;
```

```
int hasAntennaSideAreaRatio() const;
int hasAntennaDiffSideAreaRatio() const;
int hasAntennaDiffSideAreaRatioPWL() const;
int hasAntennaCumSideAreaRatio() const;
int hasAntennaCumDiffSideAreaRatio() const;
int hasAntennaCumDiffSideAreaRatioPWL() const;
int hasAntennaSideAreaFactor() const;
int hasAntennaSideAreaFactorDUO() const;
int hasAntennaCumRoutingPlusCut() const;
int hasAntennaGatePlusDiff() const;
int hasAntennaAreaMinusDiff() const;
int hasAntennaAreaDiffReducePWL() const;

char* antennaOxide() const;
double antennaAreaRatio() const;
double antennaDiffAreaRatio() const;
lefiAntennaPWL* antennaDiffAreaRatioPWL() const;
double antennaCumAreaRatio() const;
double antennaCumDiffAreaRatio() const;
lefiAntennaPWL* antennaCumDiffAreaRatioPWL() const;
double antennaAreaFactor() const;
double antennaSideAreaRatio() const;
double antennaDiffSideAreaRatio() const;
lefiAntennaPWL* antennaDiffSideAreaRatioPWL() const;
double antennaCumSideAreaRatio() const;
double antennaCumDiffSideAreaRatio() const;
lefiAntennaPWL* antennaCumDiffSideAreaRatioPWL() const;
double antennaSideAreaFactor() const;
double antennaGatePlusDiff() const;
double antennaAreaMinusDiff() const;
lefiAntennaPWL* antennaAreaDiffReducePWL() const; };
```

lefiAntennaPWL

Retrieves antenna Piece-wise Linear Format (PWL) data from a `LAYER` section of the LEF file.

For syntax information about the LEF `LAYER` sections, see "Layer (Cut)," and "Layer (Routing)" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiAntennaPWL {
     int numPWL() const;
     double PWLdiffusion(int index):
     double PWLratio(int index); };
```

lefiInfluence

Retrieves influence rule information from a `LAYER` (Routing) section of the LEF file.

For syntax information about the LEF `LAYER` (Routing) section, see "Layer (Routing)" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiInfluence {
     int numInfluenceEntry() const;
     double width(int index) const;
     double distance(int index) const;
     double spacing(int index) const; };
```

lefiLayer

Retrieves data from a `LAYER` section of the LEF file. This callback can be used for all layer types (cut, masterslice, implant, and routing). However, most of these functions apply to routing layers. Comments in the C++ syntax indicate those arguments that apply only to a particular layer type. All other arguments apply to all layer types.

For syntax information about the LEF `LAYER` sections, see "Layer (Cut)," "Layer (Masterslice or Overlap)," and "Layer (Routing)" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiLayer {
     int hasType() const;
     int hasPitch() const;                                   // Routing
     int hasXYPitch() const;
     int hasOffset() const;                                  // Routing
     int hasXYOffset() const;
     int hasWidth() const;                                   // Routing
     int hasArea() const;
     int hasDiagPitch() const;
     int hasXYDiagPitch() const;
     int hasDiagWidth() const;
     int hasDiagSpacing() const;
     int hasSpacingNumber() const;
     int hasSpacingName(int index) const;
     int hasSpacingLayerStack(int index) const;
     int hasSpacingAdjacent(int index) const;
     int hasSpacingCenterToCenter(int index) const;
     int hasSpacingRange(int index) const;                   // Routing
     int hasSpacingRangeUseLengthThreshold(int index) const;
```

```
int hasSpacingRangeInfluence(int index) const;
int hasSpacingRangeInfluenceRange(int index) const;
int hasSpacingRangeRange(int index) const;
int hasSpacingLengthThreshold(int index) const;          // Routing
int hasSpacingLengthThresholdRange(int index) const;     // Routing
int hasSpacingParallelOverlap(int index) const;
int hasSpacingArea(int index) const;
int hasSpacingEndOfLine(int index) const;
int hasSpacingParellelEdge(int index) const;
int hasSpacingTwoEdges(int index) const;
int hasSpacingAdjacentExcept(int index) const;
int hasSpacingSamenet(int index) const;
int hasSpacingSamenetPGonly(int index) const;
int hasSpacingNotchLength(int index) const;
int hasSpacingEndOfNotchWidth(int index) const;
int hasDirection() const;                                // Routing
int hasResistance() const;                               // Routing
int hasResistanceArray() const;
int hasCapacitance() const;                              // Routing
int hasCapacitanceArray() const;
int hasHeight() const;                                 // Routing
int hasThickness() const;                              // Routing
int hasWireExtension() const;                          // Routing
int hasShrinkage() const;                              // Routing
int hasCapMultiplier() const;                          // Routing
int hasEdgeCap() const;                                // Routing
int hasAntennaLength() const;                          // Routing
int hasAntennaArea() const;                            // Routing
int hasCurrentDensityPoint() const;
int hasCurrentDensityArray() const;
int hasAccurrentDensity() const;
int hasDccurrentDensity() const;

int numProps() const;
const char* propName(int index) const;
const char* propValue(int index) const;
double propNumber(int index) const;
const char propType(int index) const;
int propIsNumber(int index) const;
int propIsString(int index) const;

int numSpacing() const;                                // Cut and Routing

char* name() const;
const char* type() const;
double pitch() const;                                  // Routing
double pitchX() const;
double pitchY() const;
double offset() const;                                 // Routing
double offsetX() const;
double offsetY() cont;
```

```
double width() const;
double area() const;
double diagPitch() const;
double diagPitchX() const;
double diagPitchY() const;
double diagWidth() const;
double diagSpacing() const;
double spacing(int index) const;
char*  spacingName(int index) const;                  // Cut
int spacingAdjacentCuts(int index) const;             // Cut
double spacingAdjacentWithin(int index) const;        // Cut
double spacingArea(int index) const;                  // Cut
double spacingRangeMin(int index) const;
double spacingRangeMax(int index) const;
double spacingRangeInfluence(int index) const;
double spacingRangeInfluenceMin(int index) const;
double spacingRangeInfluenceMax(int index) const;
double spacingRangeRangeMin(int index) const;
double spacingRangeRangeMax(int index) const;
double spacingLengthThreshold(int index) const;
double spacingLengthThresholdRangeMin(int index) const;
double spacingLengthThresholdRangeMax(int index) const;

double spacingEolWidth(int index) const;
double spacingEolWithin(int index) const;
double spacingParSpace(int index) const;
double spacingParWithin(int index) const;

double spacingNotchLength(int index) const;
double spacingEndOfNotchWidth(int index) const;
double spacingEndOfNotchSpacing(int index) const;
double spacingEndOfNotchLength(int index) const;

int numMinimumcut() const;
int minimumcut(int index) const;
double minimumcutWidth(int index) const;
int hasMinimumcutWithin(int index) const;
double minimumcutWithin(int index) const;
int hasMinimumcutConnection(int index) const;      // FROMABOVE | FROMBELOW
const char* minimumcutConnection(int index) const; // FROMABOVE | FROMBELOW
int hasMinimumcutNumCuts(int index) const;
double minimumcutLength(int index) const;
double minimumcutDistance(int index) const;

const char* direction() const;                        // Routing
double resistance() const;                            // Routing
double capacitance() const;                           // Routing
double height() const;                                // Routing
double wireExtension() const;                         // Routing
double thickness() const;                             // Routing
double shrinkage() const;                             // Routing
```

```
double capMultiplier() const;                          // Routing
double edgeCap() const;                                // Routing
double antennaLength() const;                          // Routing
double antennaArea() const;                            // Routing
double currentDensityPoint() const;
void currentDensityArray(int* numPoints, double** widths,
     double** current) const;
void capacitanceArray(int* numPoints, double** widths,
     double** resValues) const;
void resistanceArray(int* numPoints, double** widths,
     double** capValues) const;                       // Routing

int numAccurrentDensity() const;
lefiLayerDensity* accurrent(int index) const;
int numDccurrentDensity() const;
lefiLayerDensity* dccurrent(int index) const;

int numAntennaModel() const;
lefiAntennaModel* antennaModel(int index) const;

int hasSlotWireWidth() const;
int hasSlotWireLength() const;
int hasSlotWidth() const;
int hasSlotLength() const;
int hasMaxAdjacentSlotSpacing() const;
int hasMaxCoaxialSlotSpacing() const;
int hasMaxEdgeSlotSpacing() const;
int hasSplitWireLength() const;
int hasMinimumDensity() const;
int hasMaximumDensity() const;
int hasDensityCheckWindow() const;
int hasDensityCheckStep() const;
int hasFillActiveSpacing() const;
int hasMaxwidth() const;
int hasMinwidth() const;
int hasMinstep() const;
int hasProtrusion() const;

double slotWireWidth() const;
double slotWireLength() const;
double slotWidth() const;
double slotLength() const;
double maxAdjecentSlotSpacing() const;
double maxCoaxialSlotSpacing() const;
double maxEdgeSlotSpacing() const;
double splitWireLength() const;
double minimumDensity() const
double maximumDensity() const;
double densityCheckWindowLength() const;
double densityCheckWindowWidth() const;
double densityCheckStep() const;
```

```
double fillActiveSpacing() const;
double maxwidth() const;
double minwidth() const;
double minstep() const;
double protrusionWidth1() const;
double protrusionLength() const;
double protrusionWidth2() const;

int numMistep() const;
double minstep(int index) const;
int hasMinstepType(int index) const;
char* minstepType(int index) const;
int hasMinstepLengthsum(int index) const;
double minstepLengthsum(int index) const;
int hasMinstepMaxedges(int index) const;
int minstepMaxedges(int index) const;

int numMinenclosedarea() const;
double minenclosedarea(int index) const;
int hasMinenclosedareaWidth(int index) const;
double minenclosedareaWidth(int index) const;

int numSpacingTable();
lefiSpacingTable* spacingTable(int index);

int numEnclosure() const;
int hasEnclosureRule(int index) const;
char* enclosureRule (int index);
double enclosureOverhang1(int index) const;
double enclosureOverhang2(int index) const;
int hasEnclosureWidth(int index) const;
double enclosureMinWidth(int index) const;
int hasEnclosureExceptExtraCut(int index) const;
double enclosureExceptExtraCut(int index) const;
int hasEnclosureMinLength(int index) const;
double enclosureMinLength(int index) const;
int numPreferEnclosure() const;
int hasPreferEnclosureRule(int index) const;
char* preferEnclosureRule(int index) const;
double preferEnclosureOverhang1(int index) const;
double preferEnclosureOverhang2(int index) const;
int hasPreferEnclosureWidth(int index) const;
double preferEnclosureMinWidth(int index) const;
int hasResistancePerCut() const;
double resistancePerCut() const;
int hasMinEdgeLength() const;
double minEdgeLength() const;
int hasDiagMinEdgeLength() const;
double diagMinEdgeLength() const;
int hasMinSize() const;
int numMinSize() const;
```

```
double minSizeWidth(int index) const;
double minSizeLength(int index)const ;

int hasMaxFloatingArea() const;
double maxFloatingArea() const;
int hasArraySpacing() const;
int hasLongArray() const;
int hasViaWidth() const;
double viaWidth() const;
double cutSpacing() const;
int numArrayCuts() const;
int arrayCuts(int index) const;
double arraySpacing(int index) const;
int hasSpacingTableOrtho() const;
lefiOrthogonal *orthogonal() const;

int hasMask() const;          // Check the layer has color mask assigned or not.
int mask() const; };          // Return the color mask number of the layer.
```

lefiLayerDensity

Retrieves data from the LAYERDENSITY statement in a LAYER section of the LEF file.

For syntax information about the LEF LAYER sections, see "Layer (Cut)," and "Layer (Routing)" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiLayerDensity {
       char* type();
       int hasOneEntry();
       double oneEntry();
       int numFrequency();
       double frequency(int index);
       int numWidths();
       double width(int index);
       int numTableEntries();
       double tableEntry(int index);
       int numCutareas();
       double cutArea(int index); };
```

lefiOrthogonal

Retrieves orthogonal spacing information from a LAYER section of the LEF file.

For syntax information about the LEF LAYER sections, see "Layer (Cut) in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiOrthogonal {
     int  numOrthogonal() const;
     double cutWithin(int index) const;
     double orthoSpacing(int index) const; };
```

lefiParallel

Retrieves parallel run length information from a LAYER (Routing) section of the LEF file.

For syntax information about the LEF LAYER (Routing) section, see "Layer (Routing)" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiParallel {
     int numLength() const;
     int numWidth() const;
     double length(int iLength) const;
     double width(int iWidth) const;
     double widthSpacing(int iWidth, int iWidthSpacing) const; };
```

lefiSpacingTable

Retrieves spacing table information from a LAYER (Routing) section of the LEF file.

For syntax information about the LEF LAYER (Routing) section, see "Layer (Routing)" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiSpacingTable {
     int isInfluence() const;
     lefiInfluence* influence() const;
     int isParallel() const;
     lefiParallel* parallel() const;
     lefiTwoWidths* twoWidths() const; };
```

lefiTwoWidths

Retrieves two-width spacing information from a LAYER (Routing) section of the LEF file.

For syntax information about the LEF LAYER (Routing) section, see "Layer (Routing)" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiTwoWidths {
     int numWidth() const;
     double width(int iWidth) const;
     int hasWidthPRL(int iWidth) const;
     double widthPRL(int iWidth) const;
     int numWidthSpacing(int iWidth) const;
     double widthSpacing(int iWidth, int iWidthSpacing) const; };
```

Macro Data Classes

The LEF MACRO data routines include the following LEF classes:

lefiDensity

Retrieves density information from the MACRO section of the LEF file.

For syntax information about the MACRO section, see "Macro" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiDensity {
     int numLayer() const;
     char* layerName(int index) const;
```

```
    int numRects(int index) const;
    struct lefiGeomRect getRect(int index, int rectIndex) const;
    double densityValue(int index, int rectIndex) const; };
```

lefiMacro

Retrieves data from the `MACRO` section of the LEF file.

For syntax information about the `MACRO` section, see "Macro" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiMacro {
    int hasClass() const;
    int hasGenerator() const;
    int hasGenerate() const;
    int hasPower() const;
    int hasOrigin() const;
    int hasEEQ() const;
    int hasLEQ() const;
    int hasSource() const;
    int hasXSymmetry() const;
    int hasYSymmetry() const;
    int has90Symmetry() const;
    int hasSiteName() const;
    int hasSitePattern() const;
    int hasSize() const;
    int hasForeign() const;
    int hasForeignOrigin(int index = 0) const;
    int hasForeignOrient(int index = 0) const;
    int hasForeignPoint(int index = 0) const;
    int hasClockType() const;
    int isBuffer() const;
    int isInverter() const;

    int numSitePattern() const;
    int numProperties() const;
    const char* propName(int index) const;
    const char* propValue(int index) const;
    double propNum(int index) const;
    const char propType(int index) const;
    int propIsNumber(int index) const;
    int propIsString(int index) const;

    const char* name() const;
    const char* macroClass() const;
    const char* generator() const;
    const char* EEQ() const;
```

```
    const char* LEQ() const;
    const char* source() const;
    const char* clockType() const;
    double originX() const;
    double originY() const;
    double power() const;
    void generate(char** name1, char** name2) const;
    lefiSitePattern* sitePattern(int index) const;
    const char* siteName() const;
    double sizeX() const;
    double sizeY() const;
    int numForeigns() const;
    int foreignOrient(int index = 0) const;   //optional - for information, see
                                              //Orientation Codes on page 21
    const char* foreignOrientStr(int index = 0) const;
    double foreignX(int index = 0) const;
    double foreignY(int index = 0) const;
    const char* foreignName(int index = 0) const; };
```

lefiMacroForeign

Retrieves data for in-place processing of a `MACRO FOREIGN` statement.

C++ Syntax

```
class lefiMacroForeign {
    public:
             lefiMacroForeign(const char *name,
                              int        hasPts,
                              double     x,
                              double     y,
                              int        hasOrient,
                              int        orient);

      const char *cellName() const;
      int        cellHasPts() const;
      double     px() const;
      double     py() const;
      int        cellHasOrient() const;
      int        cellOrient() const;

    protected:
      const char *cellName_;
      int        cellHasPts_;
      double     px_;
      double     py_;
```

```
    int        cellHasOrient_;
    int        cellOrient_;
  };
```

lefiMacroSite

Retrieves data for in-place processing of a `MACRO SITE` statement.

C++ Syntax

```
class lefiMacroSite {
     public:
                 lefiMacroSite(const char *name, const lefiSitePattern* pattern);

       const char            *siteName() const;
       const lefiSitePattern *sitePattern() const;

     protected:
       const char            *siteName_;
       const lefiSitePattern *sitePattern_;
     };
```

lefiPoints

Returns the X and Y points for the `ORIGIN` and `SIZE` statements in the `MACRO` section.

C++ Syntax

```
struct lefiPoints {
     double x;
     double y; };

typedef struct lefiPoints lefiNum;
```

Macro Examples

The following example shows a callback routine with the type `lefrMacroBeginCbkType`, and the class `const char*`.

```
int macroBeginCB (lefrCallbackType_e type,
                       const char *macroName,
                       lefiUserData userData) {
```

```
     // Incorrect type was passed in, expecting the type
     // lefiMacroBeginCbkType
         if (type != lefiMacroBeginCbkType) {
             printf("Type is not lefiMacroBeginCbkType, terminate
               parsing.\n");
             return 1;
         }

    // Expect a non null char* macroName
         if (!macroName || !*macroName) {
             printf("Macro name is null, terminate parsing.\n");
             return 1;
         }

    // Write out the macro name
         printf("Macro name is %s\n", macroName);
         return 0;}
```

The following example shows a callback routine with the type `lefrMacroCbkType`, and the class `lefiMacro`. This example only shows how to retrieve part of the data from the `lefiMacro` class.

```
int macroCB (lefrCallbackType_e type,
                  lefiMacro *macroInfo,
                  lefiUserData userData) {

         int              propNum, i, hasPrtSym = 0;
         lefiSitePattern* pattern;

    // Check if the type is correct
         if (type != lefrMacroCbkType) {
             printf("Type is not lefrMacroCbkType, terminate
               parsing.\n");
             return 1;
         }

         if (macroInfo->hasClass())
            printf("  CLASS %s\n", macroInfo->macroClass());

    if (macroInfo->hasXSymmetry()) {
            printf("  SYMMETRY X ");
            hasPrtSym = 1;
         }
         if (macroInfo->hasYSymmetry()) {   // print X Y & R90 in one line

            if (!hasPrtSym) {               // the line has not started yet
               printf("  SYMMETRY Y ");
```

```
            hasPrtSym = 1;
         }
         else                             // the line has already started
            printf("Y ");
      }
      if (macroInfo->has90Symmetry()) {
         if (!hasPrtSym) {                // the line has not started yet
            printf("  SYMMETRY R90 ");
            hasPrtSym = 1;
         }
         else                             // the line has already started
            printf("R90 ");
      }
      if (hasPrtSym) {
          printf ("\n");
          hasPrtSym = 0;
      }

// Check if SITE pattern is defined in the macro
      if (macroInfo->hasSitePattern()) {
          for (i = 0; i < macroInfo->numSitePattern(); i++ ) {
              pattern = macroInfo->sitePattern(i);
              printf("  SITE %s %g %g %d DO %g BY %g STEP %g %g\n",
                     pattern->name(), pattern->x(), pattern->y(),
                     pattern->orient(), pattern->xStart(),
                     pattern->yStart(),
                     pattern->xStep(), pattern->yStep());
          }
      }

// Check if PROPERTY is defined in the macro
      propNum = macroInfo->numProperties();
      if (propNum > 0) {
          printf("  PROPERTY ");
          for (i = 0; i < propNum; i++) {
              // value can either be a string or number
              if (macroInfo->propValue(i)) {
                  printf("%s %s ", macroInfo->propName(i),
                         macroInfo->propValue(i));
              }
              else
                  printf("%s %g ", macroInfo->propName(i),
                         macroInfo->propNum(i));
          }
          printf("\n");

      }
      return 0;}
```

Macro Obstruction Class

The LEF Macro Obstruction routines include the following LEF class:

- lefiObstruction on page 74

lefiObstruction

Retrieves data from the Macro Obstruction (`OBS`) statement in the `MACRO` section of the LEF file. The Macro Obstruction statement defines sets of obstructions (blockages) on the macro.

For syntax information about the Macro Obstruction statement, see "Macro Obstruction Statement" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiObstruction {
     lefiGeometries* geometries() const;};
```

Macro Obstruction Examples

The following example shows a callback routine with the type `lefrObstructionCbkType`, and the class `lefiObstruction`.

```
int macroObsCB (lefrCallbackType_e type,
     lefiObstruction* obsInfo,
     lefiUserData userData) {

     lefiGeometries*       geometry;
     int                   numItems;
     int                   i, j;
     lefiGeomPath*         path;
     lefiGeomPathIter*     pathIter;
     lefiGeomRect*         rect;
     lefiGeomRectIter*     rectIter;
     lefiGeomPolygon*      polygon;
     lefiGeomPolygonIter*  polygonIter;
     lefiGeomVia*          via;
     lefiGeomViaIter*      viaIter;

     // Check if the type is correct
         if (type != lefrObstructionCbkType) {
             printf("Type is not lefrObstructionCbkType,
```

```
                terminate parsing.\n");
        return 1;}

    printf("OBS\n");
    geometry = obs->geometries();
    numItems = geometry->numItems();

    for (i = 0; i < numItems; i++) {
        switch (geometry->itemType(i)) {
            case lefiGeomClassE:
                 printf("   CLASS %s\n", geometry->getClass(i));
                 break;
            case lefiGeomLayerE:
                 printf("   LAYER %s\n", geometry->getLayer(i));
                 break;
            case lefiGeomWidthE:
                 printf("   WIDTH %g\n", geometry->getWidth(i))
                 break;
            case lefiGeomPathE:
                 path = geometry->getPath(i);
                 printf("   PATH");
                 for (j = 0; j < path->numPoints; j++)
                     printf(" ( %g %g )", path->x[j], path->y[j]);
                 printf("\n");
                 break;
            case lefiGeomPathIterE:
                 pathIter = geometry->getPathIter(i);
                 printf("   PATH ITERATED");
                 for (j = 0; j < pathIter->numPoints; j++)
                     printf(" ( %g %g )", pathIter->x[j],
                             pathIter ->y[j]);
                 printf("\n");
                 printf("   DO %g BY %g STEP %g %g\n",
                         pathIter->xStart, pathIter->yStart,
                         pathIter->xStep, pathIter->yStep);
                 break;
            case lefiGeomRectE:
                 rect = geometry->getRect(i);
                 printf("   RECT ( %g %g ) ( %g %g )\n", rect->xl,
                        rect->yl, rect->xh, rect->yh);
                 break;
            case lefiGeomRectIterE:
                 rectIter = geometry->getRectIter(i);
                 printf("   RECT ITERATE ( %g %g ) ( %g %g )\n",
                         rectIter->xl, rectIter->yl,
                         rectIter->xh, rectIter->yh);
                 printf("   DO %g BY %g STEP %g %g\n",
                         rectIter->xStart, rectIter->yStart,
                         rectIter->xStep, rectIter->yStep);
                 break;
```

```
            case lefiGeomPolygonE:
                 polygon = geometry->getPolygon(i);
                 printf("   POLYGON");
                 for (j = 0; j < polygon->numPoints; j++)
                     printf(" ( %g %g )", polygon->x[j], polygon-
                         >y[j]);
                 printf("\n");
                 break;
            case lefiGeomPolygonIterE:
                 polygonIter = geometry->getPolygonIter(i);
                 printf("   POLYGON ITERATE");
                 for (j = 0; j < polygonIter->numPoints; j++)
                    printf(" ( %g %g )", polygonIter->x[j],
                         polygonIter->y[j]);
                 printf("\n");
                 printf("   DO %g BY %g STEP %g %g\n",
                         polygonIter->xStart, polygonIter->yStart,
                         polygonIter->xStep, polygonIter->yStep);
                 break;
            case lefiGeomViaE:
                 via = geometry->getVia(i);
                 printf("   VIA ( %g %g ) %s\n", via->x,
                         via->y, via->name);
                 break;
            case lefiGeomViaIterE:
                 viaIter = geometry->getViaIter(i);
                 printf("   VIA ITERATE ( %g %g ) %s\n",
                         viaIter->x, viaIter->y, viaIter->name);
                 printf("   DO %g BY %g STEP %g %g\n",
                         viaIter->xStart, viaIter->yStart,
                         viaIter->xStep, viaIter->yStep);
                 break;
        }
    }
    return 0; }
```

Macro Pin Classes

The LEF Macro Pin routines include the following LEF classes:

- lefiPin on page 77
- lefiPinAntennaModel on page 79
- lefiGeometries on page 80

lefiPin

Retrieves data from the PIN statement in the MACRO section of the LEF file. MACRO PIN statements are included in the LEF file for each macro.

For syntax information about the Macro Pin statement, see "Macro Pin Statement" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiPin {
     int hasForeign() const;
     int hasForeignOrient(int index = 0) const;
     int hasForeignPoint(int index = 0) const;
     int hasLEQ() const;
     int hasDirection() const;
     int hasUse() const;
     int hasShape() const;
     int hasMustjoin() const;
     int hasOutMargin() const;
     int hasOutResistance() const;
     int hasInMargin() const;
     int hasPower() const;
     int hasLeakage() const;
     int hasMaxload() const;
     int hasMaxdelay() const;
     int hasCapacitance() const;
     int hasResistance() const;
     int hasPulldownres() const;
     int hasTieoffr() const;
     int hasVHI() const;
     int hasVLO() const;
     int hasRiseVoltage() const;
     int hasFallVoltage() const;
     int hasRiseThresh() const;
     int hasFallThresh() const;
     int hasRiseSatcur() const;
     int hasFallSatcur() const;
     int hasCurrentSource() const;
     int hasTables() const;
     int hasAntennaSize() const;
     int hasAntennaMetalArea() const;
     int hasAntennaMetalLength() const;
     int hasAntennaPartialMetalArea() const;
     int hasAntennaPartialMetalSideArea() const;
     int hasAntennaPartialCutArea() const;
     int hasAntennaDiffArea() const;
     int hasAntennaModel() const;
     int hasTaperRule() const;
```

```
int hasRiseSlewLimit() const;
int hasFallSlewLimit() const;
int hasNetExpr() const;
int hasSupplySensitivity() const;
int hasGroundSensitivity() const;

const char* name() const;

int numPorts() const;
lefiGeometries* port(int index) const;

int numForeigns() const;
const char* foreignName(int index = 0) const;
const char* taperRule() const;
int foreignOrient(int index = 0) const;  // optional - for information, see
                                          // Orientation Codes on page 21
const char* foreignOrientStr(int index = 0) const;
double foreignX(int index = 0) const;
double foreignY(int index = 0) const;
const char* LEQ() const;
const char* direction() const;
const char* use() const;
const char* shape() const;
const char* mustjoin() const;
double outMarginHigh() const;
double outMarginLow() const;
double outResistanceHigh() const;
double outResistanceLow() const;
double inMarginHigh() const;
double inMarginLow() const;
double power() const;
double leakage() const;
double maxload() const;
double maxdelay() const;
double capacitance() const;
double resistance() const;
double pulldownres() const;
double tieoffr() const;
double VHI() const;
double VLO() const;
double riseVoltage() const;
double fallVoltage() const;
double riseThresh() const;
double fallThresh() const;
double riseSatcur() const;
double fallSatcur() const;
double riseSlewLimit() const;
double fallSlewLimit() const;
const char* currentSource() const;
const char* tableHighName() const;
const char* tableLowName() const;
```

```
int numAntennaSize() const;
double antennaSize(int index) const;
const char* antennaSizeLayer(int index) const;

int numAntennaMetalArea() const;
double antennaMetalArea(int index) const;
const char* antennaMetalAreaLayer(int index) const;

int numAntennaMetalLength() const;
double antennaMetalLength(int index) const;
const char* antennaMetalLengthLayer(int index) const;

int numAntennaPartialMetalArea() const;
double antennaPartialMetalArea(int index) const;
const char* antennaPartialMetalAreaLayer(int index) const;

int numAntennaPartialMetalSideArea() const;
double antennaPartialMetalSideArea(int index) const;
const char* antennaPartialMetalSideAreaLayer(int index) const;

int numAntennaPartialCutArea() const;
double antennaPartialCutArea(int index) const;
const char* antennaPartialCutAreaLayer(int index) const;

int numAntennaDiffArea() const;
double antennaDiffArea(int index) const;
const char* antennaDiffAreaLayer(int index) const;

const char* netExpr() const;
const char* supplySensitivity() const;
const char* groundSensitivity() const;

int numantennaModel() const;
lefiPinAntennaModel* antennaModel(int index) const;

int numProperties() const;
const char* propName(int index) const;
const char* propValue(int index) const;
double propNum(int index) const;
const char propType(int index) const;
int propIsNumber(int index) const;
int propIsString(int index) const; };
```

lefiPinAntennaModel

Retrieves antenna model information from Macro Pin statement of the LEF file.

For syntax information about the Macro Pin statement, see "Macro Pin Statement" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiAntennaModel {
     int hasAntennaGateArea() const;
     int hasAntennaMaxAreaCar() cons
     int hasAntennaMaxSideAreaCar() const;
     int hasAntennaMaxCutCar() const;

     char* antennaOxide() const;

     int numantennaGateArea() const;
     double antennaGateArea(int index) const;
     const char* antennaGateAreaLayer(int index) const;

     int numAntennaMaxAreaCar() const;
     double antennaMaxAreaCar(int index) const;
     const char* antennaMaxAreaCarLayer(int index) const;

     int numAntennaMaxSideAreaCar() conts;
     double antennaMaxSideAreaCar(int index) const;
     const char* antennaMaxSideAreaCarLayer(int index) const;

     int numAntennaMaxCutCar() const;
     double antennaMaxCutCar(int index) const;
     const char* antennaMaxCutCarLayer(int index) const; };
```

lefiGeometries

Retrieves data from the Macro Pin statement and from the Macro Obstruction statement in the `MACRO` section of the LEF file. These statements specify the pin port and obstruction geometries for the macro.

For syntax information about LEF geometries, see "Layer Geometries" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiGeometries {
     int    numItems() const;
     enum   lefiGeomEnum itemType(int index) const;
     struct lefiGeomRect* getRect(int index) const;
     struct lefiGeomRectIter* getRectIter(int index) const;
     struct lefiGeomPath* getPath(int index) const;
```

```
struct lefiGeomPathIter* getPathIter(int index) const;
int    hasLayerExceptPgNet(int index) const;
char*  getLayer(int index) const;
double getLayerMinSpacing(int index) const;
double getLayerRuleWidth(int index) const;
double getWidth(int index) const;
struct lefiGeomPolygon* getPolygon(int index) const;
struct lefiGeomPolygonIter* getPolygonIter(int index) const;
char*  getClass(int index) const;
struct lefiGeomVia* getVia(int index) const;
struct lefiGeomViaIter* getViaIter(int index) const;

int colorMask;
};
```

lefiGeomEnum

Returns the type of geometry of a macro.

C++ Syntax

```
enum lefiGeomEnum {
     lefiGeomunknown = 0,
     lefiGeomLayerE,
     lefiGeomLayerMinSpacingE,
     lefiGeomLayerRuleWidthE,
     lefiGeomWidthE,
     lefiGeomPathE,
     lefiGeomPathIterE,
     lefiGeomRectE,
     lefiGeomRectIterE,
     lefiGeomPolygonE,
     lefiGeomPolygonIterE,
     lefiGeomViaE,
     lefiGeomViaIterE,
     lefiGeomClassE,
     lefiGeomEnd };
```

lefiGeomRect

Returns data from the `RECT` statement in the `MACRO` section.

C++ Syntax

```
struct lefiGeomRect {
     double xl;
     double yl;
     double xh;
     double yh;
     int colorMask; };            //specify color mask number for the GeomRect
                                 //structure.
```

lefiGeomRectIter

Returns data from the `RECT ITERATE` statement in the `MACRO` section.

C++ Syntax

```
struct lefiGeomRectIter {
     double xl;
     double yl;
     double xh;
     double yh;
     double xStart;
     double yStart;
     double xStep;
     double yStep;
     int colorMask;};          //specify color mask number for the GeomRectIter
                              //structure.
```

Note: For the following step pattern:

```
DO numX BY numY STEP spaceX spaceY
```

The values are mapped to the structure in the following way:

Step Pattern Value	Maps to Structure Value
numX	*xStart*
numY	*yStart*
spaceX	*xStep*
spaceY	*yStep*

lefiGeomPath

Returns data from the `PATH` statement in the `MACRO` section.

C++ Syntax

```
struct lefiGeomPath {
     int numPoints;
     double* x;
     double* y;
     int colorMask; };        //specify color mask number for the GeomPath
                             //structure.
```

lefiGeomPathIter

Returns data from the `PATH ITERATE` statement in the `MACRO` section.

C++ Syntax

```
struct lefiGeomPathIter {
     int numPoints;
     double* x;
     double* y;
     double  xStart;
     double  yStart;
     double  xStep;
     double  yStep;
     int colorMask; };     //specify color mask number for the GeomPathIter
                          //structure.
```

Note: For the following step pattern:

```
DO numX BY numY STEP spaceX spaceY
```

The values are mapped to the structure in the following way:

Step Pattern Value	Maps to Structure Value
numX	*xStart*
numY	*yStart*
spaceX	*xStep*
spaceY	*yStep*

lefiGeomPolygon

Returns data from the `POLYGON` statement in the `MACRO` section.

C++ Syntax

```
struct lefiGeomPolygon {
     int numPoints;
     double* x;
     double* y;
     int colorMask; };    //specify color mask number for the GeomPolygon
                         //structure.
```

lefiGeomPolygonIter

Returns data from the `POLYGON ITERATE` statement in the `MACRO` section.

C++ Syntax

```
struct lefiGeomPolygonIter {
     int numPoints;
     double* x;
     double* y;
     double  xStart;
     double  yStart;
     double  xStep;
     double  yStep;
     int colorMask; };    //specify color mask number for the GeomPolygonIter
                         //structure.
```

Note: For the following step pattern:

```
DO numX BY numY STEP spaceX spaceY
```

The values are mapped to the structure in the following way:

Step Pattern Value	Maps to Structure Value
numX	*xStart*
numY	*yStart*
spaceX	*xStep*
spaceY	*yStep*

lefiGeomVia

Returns data from the `VIA` statement in the `MACRO` section.

C++ Syntax

```
struct lefiGeomVia {
     char*  name;
     double x;
     double y;
     int topMaskNum;       //define top mask number for the GeomVia structure.
     int cutMaskNum;       //define cut mask number for the GeomVia structure.
     int bottomMaskNum;};  //define bottom mask number for the GeomVia structure.
```

lefiGeomViaIter

Returns data from the `VIA ITERATE` statement in the `MACRO` section.

C++ Syntax

```
struct lefiGeomViaIter {
     char*  name;
     double x;
     double y;
     double xStart;
     double yStart;
     double xStep;
     double yStep;
     int topMaskNum;       //define top mask number for the GeomViaIter structure.
     int cutMaskNum;       //define cut mask number for the GeomViaIter structure.
     int bottomMaskNum;};  //define bottom mask number for the GeomViaIter
                           //structure.
```

Note: For the following step pattern:

```
DO numX BY numY STEP spaceX spaceY
```

The values are mapped to the structure in the following way:

Step Pattern Value	Maps to Structure Value
numX	*xStart*
numY	*yStart*
spaceX	*xStep*

Step Pattern Value	Maps to Structure Value
spaceY	*yStep*

Macro Pin Examples

The following example shows a callback routine with the type `lefrPinCbkType`, and the class `lefiPin`. This example only shows how to retrieve part of the data from the `lefiPin` class.

```
int macroPinCB (lefrCallbackType_e type,
                     lefiPin* pinInfo,
                     lefiUserData userData) {

         lefiGeometries*       geometry;
         int                   numPorts;
         int                   numItems;
         int                   i, j;
         lefiGeomPath*         path;
         lefiGeomPathIter*     pathIter;
         lefiGeomRect*         rect;
         lefiGeomRectIter*     rectIter;
         lefiGeomPolygon*      polygon;
         lefiGeomPolygonIter*  polygonIter;
         lefiGeomVia*          via;
         lefiGeomViaIter*      viaIter;


    // Check if the type is correct
         if (type != lefrPinCbkType) {
             printf("Type is not lefrPinCbkType, terminate parsing.\n");
             return 1;
         }


    printf("PIN %s\n", pin->name());

         if (pin->hasForeign()) {
             if (pin->hasForeignOrient())
                 printf("  FOREIGN %s STRUCTURE ( %g %g ) %d\n",
                         pin->foreignName(), pin->foreignX(),
                         pin->foreignY(), pin->foreignOrient());
             else if (pin->hasForeignPoint())
                 printf("  FOREIGN %s STRUCTURE ( %g %g )\n",
                         pin->foreignName(), pin->foreignX(),
                         pin->foreignY());
             else
```

```
            printf("  FOREIGN %s\n", pin->foreignName());
      }

    if (pin->hasLEQ())
            printf("  LEQ %s\n", pin->LEQ());

    if (pin->hasAntennaSize()) {
            for (i = 0; i < pin->numAntennaSize(); i++) {
                printf("  ANTENNASIZE %g ", pin->antennaSize(i));
               if (pin->antennaSizeLayer(i))
                   printf("LAYER %s ", pin->antennaSizeLayer(i));
               printf("\n");
            }
      }

    numPorts = pin->numPorts();
         for (i = 0; i < numPorts; i++) {
             printf("  PORT\n");
             geometry = pin->port(i);
             // A complete example can be found on page 76.
             numItems = geometry->numItems();
             for (j = 0; j < numItems; j++) {
                 switch (geometry->itemType(j)) {
                     case lefiGeomClassE:
                          printf("   CLASS %s\n", geometry->getClass(j));
                          break;
                     case lefiGeomLayerE:
                          printf("   LAYER %s\n", geometry->getLayer(j));
                          break;
                     case lefiGeomWidthE:
                          printf("   WIDTH %g\n", geometry->getWidth(j));
                          break;
                     case lefiGeomPathE:
                          ...
                          break;
                     case lefiGeomPathIterE:
                          ...
                          break;
                     case lefiGeomRectE:
                          rect = geometry->getRect(j);
                          printf("   RECT ( %g %g ) ( %g %g )\n", rect->xl,
                                 rect->yl, rect->xh, rect->yh);
                          break;
                     case lefiGeomRectIterE:
                          ...
                          break;
                     case lefiGeomPolygonE:
                          ...
                          break;
```

```
                case lefiGeomPolygonIterE:
                     ...
                     break;
                case lefiGeomViaE:
                     ...
                     break;
                case lefiGeomViaIterE:
                     ...
                     break;
           }
      }
 }
 return 0; }
```

Maximum Via Stack Class

The LEF `MAXSTACKVIA` routines include the following LEF class:

- lefiMaxStackVia on page 88

lefiMaxStackVia

Retrieves data from the `MAXVIASTACK` statement in the LEF file.

For syntax information about the LEF `MAXVIASTACK` statement, see "Maximum Via Stack" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiMaxStackVia {
     int maxStackVia() const;
     int hasMaxStackViaRange() const;
     const char* maxStackViaBottomLayer() const;
     const char* maxStackViaTopLayer() const; }
```

Miscellaneous Class

Miscellaneous routines include the following LEF class:

- lefiUserData on page 89

lefiUserData

The user data can be set or changed at any time with the `lefrSetUserData` and `lefrGetUserData` calls. Every callback returns the user data as the third argument.

C++ Syntax

```
lefiUserData lefrGetUserData()
```

Nondefault Rule Class

The LEF `NONDEFAULT RULE` routines include the following LEF class:

- lefiNonDefault on page 89

lefiNonDefault

Retrieves data from the `NONDEFAULTRULE` statement in the LEF file.

For syntax information about the LEF `NONDEFAULTRULE` statement, see "Nondefault Rule" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiNonDefault {
     const char* name() const;
     int hardSpacing() const;
     int numProps() const;
     const char* propName(int index) const;
     const char* propValue(int index) const;
     double propNumber(int index) const;
     const char propType(int index) const;
     int propIsNumber(int index) const;
     int propIsString(int index) const;

     int numLayers() const;
     const char* layerName(int index) const;
     int hasLayerWidth(int index) const;
     double layerWidth(int index) const;
     int hasLayerSpacing(int index) const;
     double layerSpacing(int index) const;
     int hasLayerWireExtension(int index) const;
     double layerWireExtension(int index) const;
     int hasLayerDiagWidth(int index) const;
     double layerDiagWidth(int index) const;
```

```
int numVias() const;
lefiVia* viaRule(int index) const;

int numSpacingRules() const;
lefiSpacing* spacingRule(int index) const;

int numUseVia() const;
const char* viaName(int index) const;
int numUseViaRule() const;
const char* viaRuleName(int index) const;
int numMinCuts() const;
const char* cutLayerName(int index) const;
int numCuts(int index) const; };
```

Nondefault Rule Examples

The following example shows a callback routine with the type `lefrNonDefaultCbkType`, and the class `lefiNonDefault`. This example only shows how to retrieve part of the data from the `lefiNonDefault` class. For examples of how to retrieve via and spacing data, see the Via Routines and Same-Net Spacing Routines sections.

```
int nonDefaultCB (lefrCallbackType_e type,
                       lefiNonDefault* nonDefInfo,
                       lefiUserData userData) {
         int          i;
         lefiVia*     via;
         lefiSpacing* spacing;


     // Check if the type is correct
         if (type != lefrNonDefaultCbkType) {
             printf("Type is not lefrNonDefaultCbkType, terminate
               parsing.\n");
             return 1; }

     // Print out nondefaultrule data
         printf("NONDEFAULTRULE %s\n", def->name());
         for (i = 0; i < def->numLayers(); i++) {
             printf("  LAYER %s\n", def->layerName(i));
             if (def->hasLayerWidth(i))
                 printf("    WIDTH %g\n", def->layerWidth(i));
             if (def->hasLayerSpacing(i))
                 printf("    SPACING %g\n", def->layerSpacing(i));}


     // handle via in nondefaultrule
         for (i = 0; i < def->numVias(); i++) {
             via = def->viaRule(i);
```

```
    // handle spacing in nondefaultrule
    for (i = 0; i < def->numSpacingRules(); i++) {
        spacing = def->spacingRule(i);}

  return 0;}
```

Property Definition Classes

The LEF `PROPERTYDEFINITIONS` routines include the following classes:

- lefiProp on page 91
- lefiPropType on page 91

lefiProp

Retrieves data from the `PROPERTYDEFINITIONS` statement in the LEF file. The `PROPERTYDEFINITIONS` statement lists all properties used in the LEF file. You must define properties before you refer to them in other routines in the LEF file.

For syntax information about the LEF `PROPERTYDEFINITIONS` statement, see "Property Definitions" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiProp {
     const char* string() const;
     const char* propType() const;
     const char* propName() const;
     char  dataType() const;
     int hasNumber() const;
     int hasRange() const;
     int hasString() const;
     int hasNameMapString() const;
     double number() const;
     double left() const;
     double right() const;};
```

lefiPropType

Retrieves the data type from the LEF `PROPERTYDEFINITIONs` statement in the LEF file, if the property is of type `REAL` or `INTEGER`.

For syntax information about the LEF `PROPERTYDEFINITIONS` statement, see "Property Definitions" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiPropType {
     const char propType(char* name) const; };
```

Property Definition Examples

The following example shows a callback routine with the type `lefrPropBeginCbkType`, and the class `void *`. This callback routine marks the beginning of the Property Definition section.

```
int propDefBeginCB (lefrCallbackType_e type,
                         void* dummy,
                         lefiUserData userData) {


    // Check if the type is correct
         if (type != lefrPropBeginCbkType) {
             printf("Type is not lefrPropBeginCbkType, terminate
               parsing.\n");
             return 1;
         }


    printf("PROPERTYDEFINITIONS\n");
         return 0;}
```

The following example shows a callback routine with the type `lefrPropCbkType`, and the class `lefiProp`. This callback routine will be called for each defined property definition.

```
int propDefCB (lefrCallbackType_e type,
                    lefiProp* propInfo,
                    lefiUserData userData) {


    // Check if the type is correct
         if (type != lefrPropCbkType) {
             printf("Type is not lefrPropCbkType, terminate
               parsing.\n");
             return 1;
         }


    // Check the object type of the property definition
         if (strcmp(propInfo->propType(), "library") == 0)
             printf("LIBRARY %s ", propInfo->propName());
```

```
        else if (strcmp(propInfo->propType(), "pin") == 0)
            printf("PIN %s ", propInfo->propName());
        else if (strcmp(propInfo->propType(), "macro") == 0)
            printf("MACRO %s ", propInfo->propName());
        else if (strcmp(propInfo->propType(), "via") == 0)
            printf("VIA %s ", propInfo->propName());
        else if (strcmp(propInfo->propType(), "viarule") == 0)
            printf("VIARULE %s ", propInfo->propName());
        else if (strcmp(propInfo->propType(), "layer") == 0)
            printf("LAYER %s ", propInfo->propName());
        else if (strcmp(propInfo->propType(), "nondefaultrule") == 0
            printf("NONDEFAULTRULE %s ", propInfo->propName());


    // Check the property type
        if (propInfo->dataType() == 'I')
            printf("INTEGER ");
        if (propInfo->dataType() == 'R')
            printf("REAL ");
        if (propInfo->dataType() == 'S')
            printf("STRING ");
        if (propInfo->dataType() == 'Q')
            printf("STRING ");
        if (propInfo->hasRange()) {
            printf("RANGE %g %g ", propInfo->left(), propInfo-
              >right());
        }
        if (propInfo->hasNumber())
            printf("%g ", propInfo->number());
        if (propInfo->hasString())
            printf("'%s' ", propInfo->string());

        printf("\n");

        return 0;}
```

The following example shows a callback routine with the type `lefrPropEndCbkType`, and the class `void *`. This callback routine marks the end of the Property Definition section.

```
int propDefEndCB (lefrCallbackType_e type,
                       void* dummy,
                       lefiUserData userData) {


    // Check if the type is correct
        if (type != lefrPropEndCbkType) {
            printf("Type is not lefrPropEndCbkType, terminate
              parsing.\n");
            return 1;}
```

```
        printf("END PROPERTYDEFINITIONS\n");
        return 0;}
```

Same-Net Spacing Class

The LEF `SPACING` routines include the following LEF class:

- lefiSpacing on page 94

lefiSpacing

Retrieves data from the `SPACING` statement in the LEF file.

C++ Syntax

```
class lefiSpacing {
     int hasStack() const;
     const char* name1() const;
     const char* name2() const;
     double distance() const;};
```

Same-Net Spacing Examples

The following example shows a callback routine with the type `lefrSpacingCbkType`, and the class `lefiSpacing`. This callback routine is called for each defined spacing between callback routines with the types `lefrSpacingBeginCbkType` and `lefrSpacingEndCbkType`.

```
int spacingCB (lefrCallbackType_e type,
                    lefiSpacing* spacingInfo,
                    lefiUserData userData) {


    // Check if the type is correct
        if (type != lefrSpacingCbkType) {
            printf("Type is not lefrSpacingCbkType, terminate
              parsing.\n");
            return 1;
        }


        printf("SAMENET %s %s %g ", spacingInfo->name1(),
                    spacingInfo->name2(), spacingInfo->distance());
        if (spacingInfo->hasStack())
```

```
        printf("STACK ");
    printf("\n");

    return 0; }
```

Site Classes

The LEF `SITE` routines include the following LEF class:

- lefiSite on page 95
- lefiSitePattern

lefiSite

Retrieves data from the `SITE` statement of the LEF file.

For syntax information about the LEF `SITE` statement, see "Site" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiSite {
     const char* name() const;
     int hasClass() const;
     const char* siteClass() const;
     double sizeX() const;
     double sizeY() const;
     int hasSize() const;
     int hasXSymmetry() const;
     int hasYSymmetry() const;
     int has90Symmetry() const;
     int hasRowPattern() const;
     int numSites() const;
     char* siteName(int index) const;
     int siteOrient(int index) const;
     char* siteOrientStr(int index) const; };
```

lefiSitePattern

Retrieves site pattern information from the `SITE` statement of the LEF file.

For syntax information about the LEF `SITE` statement, see "Site" in the *LEF/DEF Language Reference*.

C++ Syntax

```
lefiSitePattern {
     const char* name() const;
     int orient() const;
     const char* orientStr() const;
     double x() const;
     double y() const;
     int hasStepPattern () const;
     double xStart() const;
     double yStart() const;
     double xStep() const;
     double yStep() const; };
```

Site Examples

The following example shows a callback routine with the type `lefrSiteCbkType`, and the class `lefiSite`.

```
int siteCB (lefrCallbackType_e type,
                 lefiSite* siteInfo,
                 lefiUserData userData) {
         int hasPrtSym = 0;


    // Check if the type is correct
         if (type != lefrSiteCbkType) {
             printf("Type is not lefrSiteCbkType, terminate
               parsing.\n");
             return 1;
         }


         printf("SITE %s\n", siteInfo->name());
         if (siteInfo->hasClass())
             printf("  CLASS %s\n", siteInfo->siteClass());
         if (siteInfo->hasXSymmetry()) {
             printf("  SYMMETRY X ");
             hasPrtSym = 1; // set the flag that the keyword SYMMETRY
                has written
         }
         if (siteInfo->hasYSymmetry()) {
             if (hasPrtSym)
                 printf("Y ");
             else {         // keyword SYMMETRY has not been written yet
                 printf("  SYMMETRY Y ");
                 hasPrtSym = 1;
             }
         }
         if (siteInfo->has90Symmetry()) {
```

```
            if (hasPrtSym)
                printf("R90 ");
            else {
                printf("  SYMMETRY R90 ");
                hasPrtSym = 1;
            }
        }
        if (hasPrtSym)
            printf("\n");

        if (siteInfo->hasSize())
            printf("  SIZE %g BY %g\n", siteInfo->sizeX(),
              siteInfo->sizeY());
        printf("END %s\n", siteInfo->name());
        return 0;}
```

Units Class

The LEF `UNITS` routines include the following LEF class:

- lefiUnits on page 97

lefiUnits

Retrieves data from the `UNITS` statement of the LEF file. This statement defines the units of measure in LEF.

For syntax information about the LEF `UNITS` statement, see "Units" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiUnits {
     int hasDatabase();
     int hasCapacitance();
     int hasResistance();
     int hasTime();
     int hasPower();
     int hasCurrent();
     int hasVoltage();
     int hasFrequency();

     const char* databaseName();
     double databaseNumber();
     double capacitance();
     double resistance();
```

```
     double time();
     double power();
     double current();
     double voltage();
     double frequency();};
```

Units Examples

The following example shows a callback routine with the type `lefrUnitsCbkType`, and the class `lefiUnits`.

```
int unitsCB (lefrCallbackType_e type,
                  lefiUnits* unitInfo,
                  lefiUserData userData) {


    // Check if the type is correct
         if (type != lefrUnitsCbkType) {
             printf("Type is not lefrUnitsCbkType, terminate
               parsing.\n");
             return 1;}


    printf("UNITS\n");
         if (unitInfo->hasDatabase())
             printf("  DATABASE %s %g\n", unitInfo->databaseName(),
                     unitInfo->databaseNumber());
         if (unitInfo->hasCapacitance())
             printf("  CAPACITANCE PICOFARADS %g\n",
                     unitInfo->capacitance());
         if (unitInfo->hasResistance())
             printf("  RESISTANCE OHMS %g\n", unitInfo->resistance());
         if (unitInfo->hasPower())
             printf("  POWER MILLIWATTS %g\n", unitInfo->power());
         if (unitInfo->hasCurrent())
             printf("  CURRENT MILLIAMPS %g\n", unitInfo->current());
         if (unitInfo->hasVoltage())
             printf("  VOLTAGE VOLTS %g\n", unitInfo->voltage());
         if (unitInfo->hasFrequency())
             printf("  FREQUENCY MEGAHERTZ %g\n", unitInfo-
               >frequency());
         printf("END UNITS\n");
         return 0;};
```

Use Min Spacing Class

The LEF `USEMINSPACING` routines include the following LEF class:

- lefiUseMinSpacing on page 99

lefiUseMinSpacing

Retrieves data from the `USEMINSPACING` statement of the LEF file.

For syntax information about the LEF `USEMINSPACING` statement, see "Use Min Spacing" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiUseMinSpacing {
     const char* name() const;
     int value() const;};
```

Via Classes

The LEF `VIA` routines include the following LEF classes:

- lefiVia on page 99
- lefiViaLayer on page 101

lefiVia

Retrieves data from the `VIA` section of the LEF file.

For syntax information about the LEF `VIA` section, see "Via" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiVia {
     int hasDefault() const ;
     int hasGenerated() const;
     int hasForeign() const ;           // optional - for information, see
                                        // Orientation Codes on page 21
     int hasForeignPnt() const ;
     int hasForeignOrient() const ;
     int hasProperties() const ;
     int hasResistance() const ;
     int hasTopOfStack() const ;
```

```
int numLayers() const;
char* layerName(int layerNum) const;
int numRects(int layerNum) const;
double xl(int layerNum, int rectNum) const;
double yl(int layerNum, int rectNum) const;
double xh(int layerNum, int rectNum) const;
double yh(int layerNum, int rectNum) const;
int numPolygons(int layerNum) const;
struct lefiGeomPolygon getPolygon(int layerNum, int polyNum) const;

char* name() const ;
double resistance() const ;

int numProperties() const ;
char* propName(int index) const;
char* propValue(int index) const;
double propNumber(int index) const;
char propType(int index) const:
int propIsNumber(int index) const;
int propIsString(int index) const;
char* foreign() const;
double foreignX() const;
double foreignY() const;
int foreignOrient() const;
char* foreignOrientStr() const;

int hasViaRule() const;
const char* viaRuleName() const;
double xCutSize() const;
double yCutSize() const;
const char* botMetalLayer() const;
const char* cutLayer() const;
const char* topMetalLayer() const;
double xCutSpacing() const;
double yCutSpacing() const;
double xBotEnc() const;
double yBotEnc() const;
double xTopEnc() const;
double yTopEnc() const;
int hasRowCol() const;
int numCutRows() const;
int numCutCols() const;
int hasOrigin() const;
double xOffset() const;
double yOffset() const;
int hasOffset() const;
double xBotOffset() const;
double yBotOffset() const;
double xTopOffset() const;
double yTopOffset() const;
int hasCutPattern() const;
```

```
     const char* cutPattern() const;

          double xl,
          double yl,
          double xh,
          double yh);
          lefiGeometries* geom);
     int rectColorMask(int layerNum,
          int rectNum);
     int polyColorMask(int layerNum,
          int rectNum); };
```

lefiViaLayer

Retrieves data from the `LAYER` statement within the `VIA` section of the LEF file. The members of the C++ class and C structures correspond to elements of the `LAYER` statement in the `VIA` section.

For syntax information about the LEF `VIA` section, see "Via" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiViaLayer {
     int numRects();
     char* name();
     double xl(int index);
     double yl(int index);
     double xh(int index);
     double yh(int index);
     int numPolygons();
     struct lefiGeomPolygon* getPolygon(int index) const;

          double xl,
          double yl
          double xh
          double yn);
          lefiGeometries* geom);
     int rectColorMask(int index);
     int polyColorMask(int index); };
```

Via Examples

The following example shows a callback routine with the type `lefrViaCbkType`, and the class `lefiVia`.

```
int viaCB (lefrCallbackType_e type,
                lefiVia* viaInfo,
                lefiUserData userData) {


    int i, j;

         // Check if the type is correct
         if (type != lefrViaCbkType) {
             printf("Type is not lefrViaCbkType, terminate
               parsing.\n");
             return 1;}

         printf("VIA %s ", viaInfo->lefiVia::name());
         if (viaInfo->hasDefault())
             printf("DEFAULT\n");
         else
             printf("\n");
         if (viaInfo->hasTopOfStack())
             printf("  TOPOFSTACKONLY\n");
         if (viaInfo->hasForeign()) {
             printf("  FOREIGN %s ", viaInfo->foreign());
             if (viaInfo->hasForeignPnt()) {
                 printf("( %g %g ) ", viaInfo->foreignX(),
                         viaInfo->foreignY());
                if (viaInfo->hasForeignOrient())
                    printf("%s ", orientStr(viaInfo->foreignOrient()));
             }
             printf("\n");
         }
         if (viaInfo->hasProperties()) {
             printf("  PROPERTY ");
             for (i = 0; i < viaInfo->numProperties(); i++) {
                 printf("%s ", viaInfo->propName(i));
                 if (viaInfo->propIsNumber(i))
                     printf("%g ", viaInfo->propNumber(i));
                 if (viaInfo->propIsString(i))
                     printf("%s ", viaInfo->propValue(i));
             }
             printf("\n");
         }
         if (viaInfo->hasResistance())
             printf("  RESISTANCE %g\n", viaInfo->resistance());
         if (viaInfo->numLayers() > 0) {
             for (i = 0; i < viaInfo->numLayers(); i++) {
                 printf("  LAYER %s\n", viaInfo->layerName(i));
                 for (j = 0; j < viaInfo->numRects(i); j++)
                     printf("    RECT ( %g %g ) ( %g %g )\n",
                            viaInfo->xl(i, j), viaInfo->yl(i, j),
                            viaInfo->xh(i, j), viaInfo->yh(i, j));
              }
```

```
}
printf("END %s\n", viaInfo->name());
return 0;}
```

Via Rule Classes

The LEF `VIARULE` routines include the following LEF classes:

- lefiViaRule on page 103
- lefiViaRuleLayer on page 103

lefiViaRule

Retrieves data from the `VIARULE` and `VIARULE GENERATE` statements of the LEF file.

For syntax information about the LEF `VIARULE` and `VIARULE GENERATE` statements, see "Via Rule," and "Via Rule Generate" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiViaRule {
     int hasGenerate() const;
     int hasDefault() const;
     char* name() const;

     int numLayers() const;
     lefiViaRuleLayer* layer(int index);

     int numVias() const;
     char* viaName(int index) const;

     int numProps() const;
     const char* propName(int index) const;
     const char* propValue(int index) const;
     double propNumber(int index) const;
     const char propType(int index) const;
     int propIsNumber(int index) const;
     int propISString(int index) const; };
```

lefiViaRuleLayer

Retrieves data from the `LAYER` statement within the `VIARULE` and `VIARULE GENERATE` statements of the LEF file.

For syntax information about the LEF `VIARULE` and `VIARULE GENERATE` statements, see "Via Rule," and "Via Rule Generate" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class lefiViaRuleLayer {
     int hasDirection() const;
     int hasEnclosure() const;
     int hasWidth() const;
     int hasResistance() const;
     int hasOverhang() const;
     int hasMetalOverhang() const;
     int hasSpacing() const;
     int hasRect() const;

     char*  name() const;
     int isHorizontal() const;
     int isVertical() const;
     double enclosureOverhang1() const;
     double enclosureOverhang2() const;
     double widthMin() const;
     double widthMax() const;
     double overhang() const;
     double metalOverhang() const;
     double resistance() const;
     double spacingStepX() const;
     double spacingStepY() const;
     double xl() const;
     double yl() const;
     double xh() const;
     double yh() const; };
```

Via Rule Examples

The following example shows a callback routine with the type `lefrViaRuleCbkType`, and the class `lefiViaRule`. This example also shows how to retrieve data from the `lefiViaRuleLayer` class.

```
int viaRuleCB (lefrCallbackType_e type,
                    lefiViaRule* viaRuleInfo,
                    lefiUserData userData) {

         int               numLayers, numVias, i;
         lefiViaRuleLayer* vLayer;

         printf("VIARULE %s", viaRuleInfo->name());
         if (viaRuleInfo->hasGenerate())
             printf(" GENERATE\n");
```

```
else
    printf("\n");

numLayers = viaRuleInfo->numLayers();
    // If numLayers == 2, it is VIARULE without GENERATE and has
    // via name. If numLayers == 3, it is VIARULE with GENERATE, and
    // the 3rd layer is cut.
for (i = 0; i < numLayers; i++) {
    vLayer = viaRuleInfo->layer(i);
    printf("  LAYER %s\n", vLayer->name());
    if (vLayer->hasDirection()) {
        if (vLayer->isHorizontal())
            printf("    DIRECTION HORIZONTAL\n");
        if (vLayer->isVertical())
            printf("    DIRECTION VERTICAL\n");
    }
    if (vLayer->hasWidth())
        printf("    WIDTH %g TO %g\n", vLayer->widthMin(),
               vLayer->widthMax());
    if (vLayer->hasResistance())
        printf("    RESISTANCE %g\n", vLayer->resistance());
    if (vLayer->hasOverhang())
        printf("    OVERHANG %g\n", vLayer->overhang());
    if (vLayer->hasMetalOverhang())
        printf("    METALOVERHANG %g\n", vLayer-
           >metalOverhang());
    if (vLayer->hasSpacing())
        printf("    SPACING %g BY %g\n", vLayer->spacingStepX(),
               vLayer->spacingStepY());
    if (vLayer->hasRect())
        printf("    RECT ( %g %g ) ( %g %g )\n", vLayer->xl(),
                     vLayer->yl(), vLayer->xh(), vLayer->yh()); }

if (numLayers == 2) {    // should have vianames
    numVias = viaRuleInfo->numVias();
    if (numVias == 0)
        printf("Should have via names in VIARULE.\n");
    else {
        for (i = 0; i < numVias; i++)
            printf("  VIA %s\n", viaRuleInfo->viaName(i));
    }
}
printf("END %s\n", viaRuleInfo->name());
return 0;}
```

5

LEF Writer Callback Routines

You can use the Cadence® Library Exchange Format (LEF) writer with callback routines, or you can call one writer function at a time.

When you use callback routines, the writer creates a LEF file in the sequence shown in the following table. The writer also checks which sections are required for the file. If you do not provide a callback for a required section, the writer uses a default routine. If no default routine is available for a required section, the writer generates an error message.

Section	Required	Default Available
Version	no	no
Bus Bit Characters	no	no
Divider Character	no	no
Units	no	no
Property Definitions	no	no
Layer	yes	no
Via	yes	no
Via Rule	yes	no
Nondefault Rules	no	no
Spacing	no	no
Site	yes	no
Macro	yes	no
Extensions	no	no
End Library	yes	no

Callback Function Format

All callback functions use the following format.

```
int UserCallbackFunctions(
     lefwCallbackType_e callBackType,
     lefiUserData data)
```

Callback Type

The `callBackType` argument is a list of objects that contains a unique number assignment for each callback from the parser. This list allows you to use the same callback routine for different types of LEF data.

User Data

The data argument is a four-byte data item that you set. The LEF writer contains only user data. The user data is most often set to a pointer to the design data so that it can be passed to the routines.

Callback Types and Setting Routines

The following table lists the LEF writer callback-setting routines and the associated callback types.

LEF Information	Setting Routine	Callback Types
Bus Bit Characters	`void lefwSetBusBitCharsCbk (lefwVoidCbkFnType);`	`lefwBusBitCharsCbkType`
Clearance Measure	`void lefwSetClearanceMeasureCbk (lefwVoidCbkFnType);`	`lefwClearanceMeasureCbkType`
Divider Character	`void lefwSetDividerCharCbk (lefwVoidCbkFnType);`	`lefwDividerCharCbkType`
Extensions	`void lefwSetExtCbk (lefwVoidCbkFnType);`	`lefwExtCbkType`
End Library	`void lefwSetEndLibCbk (lefwVoidCbkFnType);`	`lefwEndLibCbkType`

LEF Information	Setting Routine	Callback Types
Layer	`void lefwSetLayerCbk (lefwVoidCbkFnType);`	`lefwLayerCbkType`
Macro	`void lefwSetMacroCbk (lefwVoidCbkFnType);`	`lefwMacroCbkType`
Manufacturing Grid	`void lefwSetManufacturingGridCbk (lefwVoidCbkFnType);`	`lefwManufacturingGridCbkType`
Nondefault Rule	`void lefwSetNonDefaultCbk (lefwVoidCbkFnType);`	`lefwNonDefaultCbkType`
Property Definitions	`void lefwSetPropDefCbk (lefwVoidCbkFnType);`	`lefwPropDefCbkType`
Site	`void lefwSetSiteCbk (lefwVoidCbkFnType);`	`lefwSiteCbkType`
Spacing	`void lefwSetSpacingCbk (lefwVoidCbkFnType);`	`lefwSpacingCbkType`
Units	`void lefwSetUnitsCbk (lefwVoidCbkFnType);`	`lefwUnitsCbkType`
Use Min Spacing	`void lefwSetUseMinSpacingCbk (lefwVoidCbkFnType);`	`lefwUseMinSpacingCbkType`
Version	`void lefwSetVersionCbk (lefwVoidCbkFnType);`	`lefwVersionCbkType`
Via	`void lefwSetViaCbk (lefwVoidCbkFnType);`	`lefwViaCbkType`
Via Rule	`void lefwSetViaRuleCbk (lefwVoidCbkFnType);`	`lefwViaRuleCbkType`

6

LEF Writer Routines

You can use the Cadence® Library Exchange Format (LEF) writer routines to create a program that outputs a LEF file. The LEF writer routines correspond to the sections in the LEF file. This chapter describes the routines listed below that you need to write a particular LEF section.

Routines	LEF File Sections
LEF Writer Setup and Control	Initialization and global variables
Bus Bit Characters	BUSBITCHARS statement
Clearance Measure	CLEARANCEMEASURE statement
Divider Character	DIVIDERCHAR statement
Extensions	Extensions statement
Layer (Cut, Masterslice, Overlap, Implant)	LAYER sections about cut, masterslice, overlap, and implant layers
Layer (Routing)	LAYER section about routing layers
Macro	MACRO section
Macro Obstruction	OBS section within a MACRO section
Macro Pin	PIN section within a MACRO section
Macro Pin Port	PORT section within a PIN section
Manufacturing Grid	MANUFACTURINGGRID statement
Maximum Via Stack	MAXVIASTACK statement
Nondefault Rule	NONDEFAULTRULE section
Property	PROPERTY statement in a VIA, VIARULE, LAYER, MACRO or NONDEFAULTRULE section
Property Definitions	PROPERTYDEFINTIONS statement
Same-Net Spacing	SPACING statement

Routines	LEF File Sections
Site	`SITE` statement
Units	`UNITS` statement
Use Min Spacing	`USEMINSPACING` statement
Version	`VERSION` statement
Via	`VIA` section
Via Rule	`VIARULE` statement
Via Rule Generate	`VIARULEGENERATE` statement

LEF Writer Setup and Control

The LEF writer setup and control routines initialize the reader and set global variables that are used by the reader. You must begin and end a LEF file with the `lefwInit` and `lefwEnd` routines. All other routines must be used between these two routines. The remaining routines described in this section are provided as utilities. For examples of the routines described, see "Setup Examples" on page 114.

All routines return `0` if successful.

lefwInit

Initializes the LEF writer. This routine must be used first.

Syntax

```
int lefwInit(
     FILE* file)
```

Arguments

file

Specifies the name of the LEF file to create.

lefwEnd

Ends the LEF file. This routine must be used last. This routine does not require any arguments.

Syntax

```
int lefwEnd()
```

lefwCurrentLineNumber

Returns the line number of the last line written to the LEF file. This routine does not require any arguments.

Syntax

```
int lefwCurrentLineNumber()
```

lefwNewLine

Writes a blank line. This routine does not require any arguments.

Syntax

```
int lefwNewLine()
```

lefwPrintError

Prints the return status of the `lefw*` routines.

Syntax

```
void lefwPrintError(
     int status)
```

Arguments

status

Specifies the non-zero integer returned by the LEF writer routines.

Setup Examples

The following examples show how to set up the writer. There are two ways to use the LEF writer:

- You call the write routines in your own sequence. The writer makes sure that some routines are called before others, but you must make sure the entire sequence is correct, and that all required sections are there.
- You write callback routines for each section, and the writer calls your callback routines in the sequence based on the *LEF/DEF Language Reference*. If a section is required, but you do not provide a callback routine, the writer issues a warning. If there is a default routine, the writer invokes the default routine with a message attached.

This manual includes examples with and without callback routines.

The following example uses the writer without callbacks.

```
int setupRoutine() {
         FILE* f;
         int   res;

     ...
     // Open the lef file for the writer to write.
         if ((f = fopen("lefOutputFileName","w")) == 0) {
             printf("Couldn't open output file '%s'\n",
            "lefOutputFileName");
             return(2);
         }

         // Initialize the writer. This routine has to call first. Call this
         // routine instead of lefwInitCbk(f) if you are not using the
         // callback routines.
         res = lefwInit(f);
         ...

         res = lefwEnd();
         ...

         fclose(f);

         return 0;
     }
```

The following example uses the writer with callbacks.

```
int setupRoutine() {
         FILE* f;
         int   res;
```

```
        int   userData = 0x01020304;

        ...
        // Open the lef file for the writer to write.
            if ((f = fopen("lefOutputFileName","w")) == 0) {
                printf("Couldn't open output file '%s'\n",
                "lefOutputFileName");
                return(2);
        }

        // Initialize the writer. This routine has to call first. Call this
        // routine instead of lefwInit() if you are using the writer with
        // callbacks.
        res = lefwInitCbk(f);

        // Set the user callback routines
            lefwSetAntennaCbk(antennaCB);
            lefwSetBusBitCharsCbk(busBitCharsCB);
            lefwSetCaseSensitiveCbk(caseSensCB);
            lefwSetCorrectionTableCbk(correctTableCB);
            lefwSetEndLibCbk(endLibCB);
            ...

        // Invoke the parser
            res = lefrWrite(f, "lefInputFileName", (void*)userData);
            if (res != 0) {
                printf("LEF writer returns an error\n");
                return(2);
        }

         fclose(f);

         return 0;
    }
```

The following example shows how to use the callback routine to mark the end of the LEF file. The type is `lefwEndLibCbkType`.

```
#define CHECK_RES(res)       \
        if (res) {              \
            lefwPrintError(res); \
            return(res);         \
        }

    int endLibCB (lefwCallbackType_e type,
                  lefiUserData userData) {
        int    res;

        // Check if the type is correct
        if (type != lefwEndLibCbkType) {
            printf("Type is not lefwEndLibCbkType, terminate
```

```
                writing.\n");
            return 1;
        }

        res = lefwEnd();
        CHECK_RES(res);
        return 0;
    }
```

Bus Bit Characters

The Bus Bit Characters routine writes a LEF BUSBITCHARS statement. The BUSBITCHARS statement is optional and can be used only once in a LEF file. For syntax information about the LEF BUSBITCHARS statement, see "Bus Bit Characters" in the *LEF/DEF Language Reference*.

The BUSBITCHARS statement is part of the LEF file header (which also includes the VERSION, and DIVIDERCHAR statements). If the statements in the header section are not defined, many applications assume default values for them. However, the default values are not formally part of the language definition; therefore you cannot be sure that the same assumptions are used in all applications. You should always explicitly define these values.

This routine returns 0 if successful.

lefwBusBitChars

Writes a BUSBITCHARS statement.

Syntax

```
int lefwBusBitChars(
     const char* busBitChars)
```

Arguments

busBitChars

Specifies the pair of characters used to specify bus bits when LEF names are mapped to or from other databases. The characters must be enclosed in double quotation marks.

Bus Bit Characters Example

The following example shows a callback routine with the type `lefwBusBitCharsCbkType`.

```
int busBitCharsCB (lefwCallbackType_e type,
                        lefiUserData userData) {
         int res;

         // Check if the type is correct
         if (type != lefwBusBitCharsCbkType) {
             printf("Type is not lefwBusBitCharsCbkType, terminate
               writing.\n");
             return 1;
         }
         res = lefwBusBitChars("<>");
         CHECK_RES(res);
         return 0;}
```

Clearance Measure

The Clearance Measure routine writes a LEF `CLEARANCEMEASURE` statement. The `CLEARANCEMEASURE` statement is optional and can be used only once in a LEF file. For syntax information about the LEF `CLEARANCEMEASURE` section, see "Clearance Measure" in the *LEF/DEF Language Reference*.

This routine returns `0` if successful.

lefwClearanceMeasure

Writes a `CLEARANCEMEASURE` statement.

Syntax

```
int lefwClearanceMeasure(
     const char* type)
```

Arguments

type

Specifies the type of clearance spacing that will be applied to obstructions (blockages) and pins in cells.
Value: Specify one of the following:

`MAXXY`	Uses the larger x and y distances for spacing between objects.
`EUCLIDEAN`	Uses euclidean distance for spacing between objects.

Divider Character

The Divider Character routine writes a LEF `DIVIDERCHAR` statement. The `DIVIDERCHAR` statement is optional and can be used only once in a LEF file. For syntax information about the LEF `DIVIDERCHAR` statement, see "Divider Character" in the *LEF/DEF Language Reference*.

The `DIVIDERCHAR` statement is part of the LEF file header (which also includes the `VERSION`, and `BUSBITCHARS` statements). If the statements in the header section are not defined, many applications assume default values for them. However, the default values are not formally part of the language definition; therefore you cannot be sure that the same assumptions are used in all applications. You should always explicitly define these values.

This routine returns `0` if successful.

lefwDividerChar

Writes a `DIVIDERCHAR` statement.

Syntax

```
int lefwDividerChar(
     const char* dividerChar)
```

Arguments

dividerChar

Specifies the character used to express hierarchy when LEF names are mapped to or from other databases. The character must be enclosed in double quotation marks.

Note: If the divider character appears in a LEF name as a regular character, you must use a backslash (\) before the character to prevent the LEF reader from interpreting the character as a hierarchy delimiter.

Divider Character Examples

The following example shows a callback routine with the type `lefwDividerCharCbkTyp`.

```
int dividerCB (lefwCallbackType_e type,
                    lefiUserData userData) {
         int    res;

         // Check if the type is correct
         if (type != lefwDividerCharCbkType) {
             printf("Type is not lefwDividerCharCbkType, terminate
               writing.\n");
             return 1;
         }

         res = lefwDividerChar(":");
         CHECK_RES(res);
         res = lefwNewLine();    // add an empty line
         CHECK_RES(res);

         return 0;}
```

Extensions

Extensions routines write a LEF `BEGINEXT` statement. The `BEGINEXT` statement is optional and can be used more than once in a LEF file.

Extensions routines let you add customized syntax to the LEF file that can be ignored by tools that do not use that syntax. You can also use extensions to add new syntax not yet supported by your version of LEF, if you are using version 5.1 or later. For syntax information about the LEF `EXTENSIONS section`, see "Extensions" in the *LEF/DEF Language Reference*.

You must begin and end a LEF `BEGINEXT` statement with the `lefwStartBeginext` and `lefwEndBeginext` routines. All LEF writer routines that define `EXTENSIONS` routines must be included between these routines.

For examples of the routines described here, see "Extensions Examples" on page 122.

All routines return `0` if successful.

lefwStartBeginext

Starts the `EXTENSIONS` statement using the specified tag.

Syntax

```
int lefwStartBeginext(
     const char* tag)
```

Arguments

`tag`

Identifies the extension block. The tag must be enclosed in double quotation marks.

lefwEndBeginext

Writes the `ENDEXT` statement.

Syntax

```
int lefwEndBeginext()
```

lefwBeginextCreator

Writes a `CREATOR` statement. The `CREATOR` statement is optional and can be used only once in an `EXTENSIONS` statement.

Syntax

```
int lefwBeginextCreator(
     const char* creator)
```

Arguments

creator
Specifies a string value that defines the creator value.

lefwBeginextDate

Writes a DATE statement that specifies the current system time and date. The DATE statement is optional and can be used only once in an EXTENSIONS statement.

Syntax

```
int lefwBeginextDate()
```

lefwBeginextRevision

Writes a REVISION statement. The REVISION statement is optional and can be used only once in an EXTENSIONS statement.

Syntax

```
int lefwBeginextRevision(
     int vers1,
     int vers2)
```

Arguments

vers1, *vers2*
Specify the values used for the revision number string.

lefwBeginextSyntax

Adds customized syntax to the LEF file. This routine is optional and can be used more than once in an EXTENSIONS statement.

Syntax

```
int lefwBeginextSyntax(
     const char* title,
     const char* string)
```

Arguments

title, *string*

Specify any values you need.

Extensions Examples

The following example shows a callback routine with the type `lefwExtCbkType`. This example only shows the usage of some functions related to array.

```
int extCB (lefwCallbackType_e type,
                lefiUserData userData) {
         int    res;

         // Check if the type is correct
         if (type != lefwExtCbkType) {
             printf("Type is not lefwExtCbkType, terminate
               writing.\n");
             return 1;
         }

         res = lefwStartBeginext("SIGNATURE");
         CHECK_RES(res);
         res = lefwBeginextCreator("CADENCE");
         CHECK_RES(res);
         res = lefwBeginextDate();
         CHECK_RES(res);
         res = lefwEndBeginext();
         CHECK_RES(res);

         return 0;}
```

Layer (Cut, Masterslice, Overlap, Implant)

The following layer routines write `LAYER` sections about cut, masterslice, overlap, and implant layers. At least one `LAYER` section is required in a LEF file, and more than one `LAYER` section is generally required to describe a layout. For syntax information about the `LAYER` sections for cut, masterslice, overlap, and implant layers, see "Layer (Cut)", "Layer (Masterslice or Overlap)", and "Layer (Implant)" in the *LEF/DEF Language Reference*.

You must begin and end a LEF `LAYER` section with the `lefwStartLayer` and `lefwEndLayer` routines. You create one `LAYER` section for each layer you need to define.

For examples of the routines described here, see "Layer Examples" on page 140.

In addition to the routines described in this section, you can include a `PROPERTY` statement in a `LAYER` section. For more information about these routines, see "Property" on page 224.

All routines return `0` if successful.

Defining Masterslice and Overlap Layers

To define a masterslice or overlap layer, you only need to use the `lefwStartLayer` and `lefwEndLayer` routines. No additional routines are required to define these layers.

Defining Cut Layers

To define a cut layer, you must use the `lefwLayerCutSpacing` routine to start the spacing and the `lefwLayerCutSpacingEnd` routine to end the spacing. These must be used between the `lefwStartLayer` and `lefwEndLayer` routines. Any other routines are optional and must be included after the `lefwLayerCutSpacing` routine.

Defining Implant Layers

To define an implant layer, you must specify the `lefwLayerWidth` routine between the `lefwStartLayer` and `lefwEndLayer` routines.

lefwStartLayer

Starts the `LAYER` section. Each cut, masterslice, overlap, and implant layer must be defined by a separate `lefwStartLayer`, `lefwEndLayer` routine pair.

Syntax

```
int lefwStartLayer(
     const char* layerName,
     const char* type)
```

Arguments

layerName

Specifies the name of the layer being defined.

type

Specifies the type of layer being defined.
Value: CUT, MASTERSLICE, OVERLAP, or IMPLANT

lefwEndLayer

Ends the LAYER section for the specified layer.

Syntax

```
int lefwEndLayer(
     const char* layerName)
```

lefwLayerACCurrentDensity

Writes an ACCURRENTDENSITY statement for a cut layer. The ACCURENTDENSITY statement is optional, and can be used only once in a LAYER section.

Syntax

```
int lefwLayerACCurrentDensity(
     const char* type,
     double value)
```

Arguments

type

Specifies one of the AC current limits, PEAK, AVERAGE, or RMS.

value

Specifies a maximum current limit for the layer in milliamps per square micron. If you specify 0, you must call the lefwLayerACFrequency and lefwLayerACTableEntries routines.

lefwLayerACCutarea

Writes a CUTAREA statement for a cut layer. The CUTAREA statement is optional if you specify a FREQUENCY statement, and can be used only once in an ACCURENTDENSITY statement.

Syntax

```
int lefwLayerACCutarea(
     int numCutareas,
     double* cutareas)
```

Arguments

numCutareas

Specifies the number of cut area values.

cutareas

Specifies the cut area values, in square microns. If you specify only one cut area value, there is no cut area dependency, and the table entries are assumed to apply to all cut areas.

lefwLayerACFrequency

Writes a `FREQUENCY` statement for a cut layer. The `FREQUENCY` statement is required if you specify a value of `0` in the `lefwLayerACCurrentDensity` routine, and can be used only once in an `ACCURENTDENSITY` statement.

Syntax

```
int lefwLayerACFrequency(
     int numFrequency,
     double* frequency)
```

Arguments

numFrequency

Specifies the number of frequency values.

frequency

Specifies the frequency values, in megahertz. If you specify only one frequency value, there is no frequency dependency, and the table entries are assumed to apply to all frequencies.

lefwLayerACTableEntries

Writes a `TABLEENTRIES` statement for a cut layer. The `TABLENTRIES` statement is required if you specify a `FREQUENCY` statement, and can be used only once in an `ACCURENTDENSITY` statement.

Syntax

```
int lefwLayerACTableEntries(
     int numEntries,
     double* entries)
```

Arguments

numEntries

Specifies the number of table entry values.

entries

Specifies the maximum cut area for each frequency and cut area pair specified in the `FREQUENCY` and `CUTAREA` statements, in milliamps per square micron.

lefwLayerAntennaAreaFactor

Writes an `ANTENNAAREAFACTOR` statement for a cut layer. The `ANTENNAAREAFACTOR` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section.

Syntax

```
int lefwLayerAntennaAreaFactor(
     double value
     const char* diffUseOnly)
```

Arguments

value

Specifies the adjust or multiply factor for the antenna metal calculation.

`diffUseOnly`

Optional argument that specifies the current antenna factor should be used only when the corresponding layer is connected to the diffusion. Specify `NULL` to ignore this argument.

lefwLayerAntennaAreaRatio

Writes an `ANTENNAAREARATIO` statement for a cut layer. The `ANTENNAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section.

Syntax

```
int lefwLayerAntennaAreaRatio(
     double value)
```

Arguments

`value`

Specifies the antenna ratio, using the bottom area of the metal wire that is not connected to the diffusion diode.

lefwLayerAntennaCumAreaRatio

Writes an `ANTENNACUMAREARATIO` statement for a cut layer. The `ANTENNACUMAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section.

Syntax

```
int lefwLayerAntennaCumAreaRatio(
     double value)
```

Arguments

`value`

Specifies the cumulative antenna ratio, using the bottom area of the metal wire that is not connected to the diffusion diode.

lefwLayerAntennaCumDiffAreaRatio

Writes an `ANTENNACUMDIFFAREARATIO` statement for a cut layer. The `ANTENNACUMDIFFAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaCumDiffAreaRatioPWL` in the same `LAYER` section.

Syntax

```
int lefwLayerAntennaCumDiffAreaRatio(
     double value)
```

Arguments

value

Specifies the cumulative antenna ratio, using the bottom area of the metal wire that is connected to the diffusion diode.

lefwLayerAntennaCumDiffAreaRatioPwl

Writes an `ANTENNACUMDIFFAREARATIOPWL` statement for a cut layer. The `ANTENNACUMDIFFAREARATIOPWL` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaCumDiffAreaRatio` in the same `LAYER` section.

Syntax

```
int lefwLayerAntennaCumDiffAreaRatioPwl(
     int numPwls,
     double diffusions,
     double ratios)
```

Arguments

numPwls

Specifies the number of diffusion-ratio pairs.

diffusions

Specifies the diffusion values.

ratios

Specifies the ratio values.

lefwLayerAntennaDiffAreaRatio

Writes an `ANTENNADIFFAREARATIO` statement for a cut layer. The `ANTENNADIFFAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaDiffAreaRatioPWL` in the same `LAYER` section.

Syntax

```
int lefwAntennaDiffAreaRatio(
     double value)
```

Arguments

value

Specifies the antenna ratio, using the bottom area of the wire that is connected to the diffusion diode.

lefwLayerAntennaDiffAreaRatioPwl

Writes an `ANTENNADIFFAREARATIOPWL` statement for a cut layer. The `ANTENNADIFFAREARATIOPWL` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaDiffAreaRatio` in the same `LAYER` section.

Syntax

```
int lefwAntennaDiffAreaRatioPWL(
     int numPwls,
     double diffusions,
     double ratios)
```

Arguments

numPwls

Specifies the number of diffusion-ratio pairs.

diffusions

Specifies the diffusion values.

ratios

Specifies the ratio values.

lefwLayerAntennaModel

Writes an ANTENNAMODEL statement for a cut layer. The ANTENNAMODEL statement is optional and can be used more than once in a LAYER section.

Syntax

```
int lefwLayerAntennaModel(
     const char* oxide)
```

Arguments

oxide

Specifies the oxide model for the layer. Each model can be specified once per layer. If you specify an ANTENNAMODEL statement, that value affects all ANTENNA* statements for the layer that follow it until you specify another ANTENNAMODEL statement.
Value: OXIDE1, OXIDE2, OXIDE3, or OXIDE4

Note: OXIDE1 and OXIDE2 are currently supported. If you specify OXIDE3 or OXIDE4, current tools parse and ignore them.

lefwLayerArraySpacing

Writes an ARRAYSPACING statement for a cut layer. The ARRAYSPACING statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerArraySpacing(
     int longArray,
     double viaWidth,
     double cutSpacing,
     int numArrayCut,
     int* arrayCuts,
     double* arraySpacings)
```

Arguments

longArray

Optional argument that indicates that the via can use *N x M* cut arrays, where *N = arrayCuts* and *M* can be any value, including one that is larger than *N*. Specify `0` to ignore this argument.

viaWidth

Optional argument that specifies the via width. The array spacing rules only apply when the via metal width is greater than or equal to *viaWidth*. Specify `0` to ignore this argument.

cutSpacing

Specifies the edge-of-cut to edge-of-cut spacing inside one cut array.

numArrayCuts

Specifies the number of *arrayCuts* and *arraySpacings* pairs provided.

arrayCuts

Specifies the size of the cut arrays.
A large via array with a size greater than or equal to *arrayCuts* x *arrayCuts* in both dimensions must use *N x N* cut arrays (where *N = arrayCuts*) separated from other cut arrays by a distance greater than or equal to *arraySpacing*.
If you specify multiple *arrayCuts* and *arraySpacings*, the *arrayCuts* values must be specified in increasing order.

arraySpacings

Specifies the spacing between the cut arrays.

lefwLayerCutSpacing

Starts a `SPACING` statement for a cut layer. Call `lefwLayerCutSpacingEnd` to end each spacing.

The `SPACING` statement is optional and can be used more than once in a `LAYER` section.

Syntax

```
int lefwLayerCutSpacing(
     double spacing)
```

Arguments

spacing

Specifies the minimum spacing allowed between via cuts, in microns.

lefwLayerCutSpacingAdjacent

Writes an ADJACENTCUTS statement for a SPACING statement for a cut layer. The ADJACENTCUTS statement is optional. You can specify only one of the following statements per spacing: LAYER, ADJACENTCUTS, AREA, or PARALLELOVERLAP.

Syntax

```
int lefwLayerCutSpacingAdjacent(
     int viaCuts,
     double distance,
     int stack)
```

Arguments

viaCuts

Optional argument that specifies the number of via cuts—either 2, 3, or 4.

distance

Specifies the distance between via cuts, in microns.

stack

Optional argument that sets the EXCEPTSAMEPGNET keword for the spacing. If this keyword is set, the ADJACENTCUTS rule does not apply between cuts if they are on the same net, and are on a power and ground net.

lefwLayerCutSpacingArea

Writes an AREA statement for a SPACING statement for a cut layer. The AREA statement is optional. You can specify only one of the following statements per spacing: LAYER, ADJACENTCUTS, AREA, or PARALLELOVERLAP.

Syntax

```
int lefwLayerCutSpacingArea(
     double cutArea)
```

Arguments

cutArea

Specifies the cut area. Any cut with an area equal to or greater than this number requires additional spacing.

lefwLayerCutSpacingCenterToCenter

Writes a `CENTERTOCENTER` statement for a `SPACING` statement for a cut layer. The `CENTERTOCENTER` statement is optional.

Syntax

```
int lefwLayerCutSpacingCenterToCenter()
```

lefwLayerCutSpacingEnd

Ends a `SPACING` statement for a cut layer.

Syntax

```
int lefwLayerCutSpacingEnd()
```

lefwLayerCutSpacingLayer

Writes a `LAYER` statement for a `SPACING` statement for a cut layer. The `LAYER` statement is optional. You can specify only one of the following statements per spacing: `LAYER`, `ADJACENTCUTS`, `AREA`, or `PARALLELOVERLAP`.

Syntax

```
int lefwLayerCutSpacingLayer(
     const char* name2,
     int stack)
```

Arguments

name2

Specifies the second layer name.

stack

Optional argument indicating that same-net cuts on two different layers can be stacked if they are exactly aligned; otherwise, the cuts must have *cutSpacing* between them. Specify 0 to ignore this argument.

lefwLayerCutSpacingParallel

Writes a PARALLELOVERLAP statement for a SPACING statement for a cut layer. The PARALLELOVERLAP statement is optional. You can specify only one of the following statements per spacing: LAYER, ADJACENTCUTS, AREA, or PARALLELOVERLAP.

Syntax

```
int lefwLayerCutSpacingParallel()
```

lefwLayerCutSpacingSamenet

Writes a SAMENET statement for a SPACING statement for a cut layer. The SAMENET statement is optional.

Syntax

```
int lefwLayerCutSpacingSameNet()
```

lefwLayerCutSpacingTableOrtho

Writes a SPACINGTABLE ORTHOGONAL statement for a cut layer. The SPACINGTABLE ORTHOGONAL statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerCutSpacingTableOrtho(
     int numSpacing,
     double* cutWithins,
     double* orthoSpacings)
```

Arguments

numSpacing

Specifies the number of *cutWithins* and *orthoSpacings* pairs provided.

cutWithins

Specifies the distance between cuts, in microns.
If two cuts have parallel overlap greater than 0 and are less than *cutWithin* distance from each other, then any other cuts in an orthogonal direction must be equal to or greater than *orthoSpacings*.

orthoSpacings

Specifies the orthogonal spacing, in microns.

lefwLayerDCCurrentDensity

Writes the DCCURRENTDENSITY statement for a cut layer. The DCCURENTDENSITY statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerDCCurrentDensity(
     const char* type,
     double value)
```

Arguments

type

Specifies the DC current limit, AVERAGE.

value

Specifies a current limit for the layer in milliamps per square microns. If you specify 0, you must call the lefwLayerDCCutarea and lefwLayerDCTableEntries routines.

lefwLayerDCCutarea

Writes a CUTAREA statement for a cut layer. The CUTAREA statement is required if you specify a value of 0 in the lefwLayerDCCurrentDensity routine, and can be used only once in a DCCURENTDENSITY statement.

Syntax

```
int lefwLayerDCCutarea(
     int numCutareas,
     double* cutareas)
```

Arguments

numCutareas

Specifies the number of cut area values.

cutareas

Specifies the cut area values, in square microns.

lefwLayerDCTableEntries

Writes a `TABLEENTRIES` statement for a cut layer. The `TABLENTRIES` statement is required if you specify a `CUTAREA` statement, and can be used only once in a `DCCURENTDENSITY` statement.

Syntax

```
int lefwLayerDCTableEntries(
     int numEntries,
     double* entries)
```

Arguments

numEntries

Specifies the number of table entry values.

entries

Specifies the maximum current density for each specified cut area, in milliamps per square micron.

lefwLayerEnclosure

Writes an `ENCLOSURE` statement for a cut layer. The `ENCLOSURE` statement is optional and can be used more than once in a `LAYER` section.

Syntax

```
lefwLayerEnclosure(
     const char* location,
     double overhang1,
     double overhang2,
     double width)
```

Arguments

location

Optional argument that specifies whether the overhang is required on the routing layers above or below the cut layer. Specify `""` to ignore this argument.
Value: `ABOVE` or `BELOW`

overhang1 overhang2

Specifies that any rectangle from this cut layer requires the routing layers to overhang by *overhang1* on two opposite sides, and by *overhang2* on the other two opposite sides.

width

Optional argument that specifies that the enclosure rule only applies when the width of the routing layer is greater than or equal to *width*. Specify `0` to ignore this argument.

lefwLayerEnclosureLength

Writes an `ENCLOSURE` statement with a `LENGTH` keyword for a cut layer. This routine lets you specify a minimum length instead of the width. The `ENCLOSURE` statement is optional and can be used more than once in a `LAYER` section.

Syntax

```
int lefwLayerEnclosureLength(
     const char* location,
     double overhang1,
     double overhang2,
     double minLength)
```

Arguments

location

Optional argument that specifies whether the overhang is required on the routing layers above or below the cut layer. If you don't specify this argument, the rule applies to both adjacent routing layers; specify `""` to ignore this argument.
Value: `ABOVE` or `BELOW`

overhang1 overhang2

Overhange values. Any rectangle from this cut layer requires the routing layers to overhang by *overhang1* on two opposite sides, and by *overhang2* on the other two opposite sides.

minLength

Optional argument that specifies that the total length of the longest opposite-side overhangs must be greater than or equal to *minLength* to make this enclosure valid. The *minLength* is measured at the center of the cut. Specify 0 to ignore this argument.

lefwLayerEnclosureWidth

Writes an ENCLOSURE statement with an EXCEPTEXTRACUT keyword for a cut layer. This routine is similar to lefwLayerEnclosure except that it lets you specify EXCEPTEXTRACUT. The ENCLOSURE statement is optional and can be used more than once in a LAYER section.

Syntax

```
int lefwLayerEnclosureWidth(
     const char* location,
     double overhang1,
     double overhang2,
     double width,
     double cutWithin)
```

Arguments

location

Optional argument that specifies whether the overhang is required on the routing layers above or below the cut layer. If you don't specify this argument, the rule applies to both adjacent routing layers; specify "" to ignore this argument.
Value: ABOVE or BELOW

overhang1 overhang2

Overhange values. Any rectangle from this cut layer requires the routing layers to overhang by *overhang1* on two opposite sides, and by *overhang2* on the other two opposite sides.

width

Optional argument that specifies that the enclosure rule only applies when the width of the routing layer is greater than or equal to *width*. Specify 0 to ignore this argument. If you do not specify this argument, the enclosure rule applies to all widths (as if *width* was 0).

cutWithin

Optional argument that sets the EXCEPTEXTRACUT *cutWithin* keyword. Specifies that if there is another via cut less than or equal to *cutWithin*, then this ENCLOSURE with `WIDTH` rule is ignored and the `ENCLOSURE` rules for minimum width wires are applied to the via cuts instead. Specify `0` to ignore this argument.

lefwLayerPreferEnclosure

Writes a `PREFERENCLOSURE` statement for a cut layer. The `PREFERENCLOSURE` statement is optional and can be used more than once in a `LAYER` section.

Note: The `PREFERENCLOSURE` statement specifies preferred enclosure rules that can improve manufacturing yield, instead of enclosure rules that absolutely must be met (`ENCLOSURE` statement).

Syntax

```
lefwLayerPreferEnclosure(
     const char* location,
     double overhang1,
     double overhang2,
     double width)
```

Arguments

location

Optional argument that specifies whether the overhang is required on the routing layers above or below the cut layer. Specify `""` to ignore this argument.
Value: `ABOVE` or `BELOW`

overhang1 overhang2

Specifies that any rectangle from this cut layer requires the routing layers to overhang by *overhang1* on two opposite sides, and by *overhang2* on the other two opposite sides. The overhang values must be equal to or larger than the overhang values in the `ENCLOSURE` rule.

width

Optional argument that specifies that the enclosure rule only applies when the width of the routing layer is greater than or equal to *width*. Specify `0` to ignore this argument.

lefwLayerResistancePerCut

Writes a `RESISTANCE` statement for the cut layer. The `RESISTANCE` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
lefwLayerResistancePerCut(
     double resistance)
```

Arguments

resistance

Specifies the resistance per cut on this layer. LEF vias without their own specific resistance value, or DEF vias from a via rule without a resistance per cut value, can use this resistance value.

lefwLayerWidth

Writes a `WIDTH` statement for an implant or a cut layer. The `WIDTH` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerWidth(
     double minWidth)
```

Arguments

minWidth

Specifies the minimum width for the layer.

Layer Examples

The following example shows a callback routine with the type `lefwLayerCbkType`. This example shows how to create a cut, masterslice, or overlap layer. For an example of a routing layer, see the Layer (Routing) section.

```
int layerCB (lefwCallbackType_e type,
                  lefiUserData userData) {
          int    res;
          double *current;

          // Check if the type is correct
          if (type != lefwLayerCbkType) {
              printf("Type is not lefwLayerCbkType, terminate
                writing.\n");
              return 1;
```

```
}

current = (double*)malloc(sizeof(double)*3);

res = lefwStartLayer("CA", "CUT");
CHECK_RES(res);
res = lefwLayerDCCurrentDensity("AVERAGE", 0);
CHECK_RES(res);
current[0] = 2.0;
current[1] = 5.0;
current[2] = 10.0;
res = lefwLayerDCWidth(3, current);
CHECK_RES(res);
current[0] = 0.6E-6;
current[1] = 0.5E-6;
current[2] = 0.4E-6;
res = lefwLayerDCTableEntries(3, current);
CHECK_RES(res);
res = lefwEndLayer("CA");
CHECK_RES(res);
free((char*)current);

res = lefwStartLayer("POLYS", "MASTERSLICE");
CHECK_RES(res);
res = lefwStringProperty("lsp", "top");
CHECK_RES(res);
res = lefwIntProperty("lip", 1);
CHECK_RES(res);
res = lefwRealProperty("lrp", 2.3);
CHECK_RES(res);
res = lefwEndLayer("POLYS");
CHECK_RES(res);

res = lefwStartLayer("OVERLAP", "OVERLAP");
CHECK_RES(res);
res = lefwEndLayer("OVERLAP");
CHECK_RES(res);

return 0;}
```

Layer (Routing)

Routing layer routines write `LAYER` sections about routing layers. At least one `LAYER` section is required in a LEF file, and more than one `LAYER` section is generally required to describe a layout. For syntax information about the `LAYER` section for routing layers, see "Layer (Routing)" in the *LEF/DEF Language Reference*.

You must begin and end a LEF LAYER section with the lefwStartLayerRouting and lefwEndLayerRouting routines. The remaining routing layer routines defined in this section must be included between these routines. You create one LAYER section for each routing layer you need to define.

For examples of the routines described here, see "Routing Layer Examples" on page 185

In addition to the routines described in this section, you can include a PROPERTY statement within a LAYER section. For more information about these routines, see "Property" on page 224.

All routines return 0 if successful.

lefwStartLayerRouting

Starts the LAYER section. The LEF writer automatically writes the TYPE ROUTING statement. This routine is required to define a routing layer and can be used more than once. Each routing layer must be defined by a separate lefwStartLayerRouting, lefwEndLayerRouting routine pair.

Syntax

```
int lefwStartLayerRouting(
     const char* layerName)
```

Arguments

layerName
Specifies the name of the routing layer being defined.

lefwEndLayerRouting

Ends the LAYER section for the specified routing layer.

Syntax

```
int lefwEndLayerRouting(
     const char* layerName)
```

lefwDensityCheckStep

Writes a DENSITYCHECKSTEP statement. The DENSITYCHECKSTEP statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwDensityCheckStep(
     double stepValue)
```

Arguments

stepValue

Specifies the stepping distance for metal density checks, in distance units.

lefwDensityCheckWindow

Writes a DENSITYCHECKWINDOW statement. The DENSITYCHECKWINDOW statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwDensityCheckWindow(
     double windowLength,
     double windowWidth)
```

Arguments

windowLength

Specifies the length of the check window, in distance units.

windowWidth

Specifies the width of the check window, in distance units.

lefwFillActiveSpacing

Writes a FILLACTIVESPACING statement. The FILLACTIVESPACING statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwFillActiveSpacing(
     double spacing)
```

Arguments

spacing

Specifies the spacing between metal fills and active geometries.

lefwLayerACCurrentDensity

Writes an `ACCURRENTDENSITY` statement. The `ACCURENTDENSITY` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerACCurrentDensity(
     const char* type,
     double value)
```

Arguments

type

Specifies the type of AC current limit.
Value: `PEAK`, `AVERAGE`, or `RMS`

value

Specifies a maximum current for the layer, in milliamps per micron. If you specify `0`, you must specify the `lefwLayerACFrequency` and `lefwLayerACTableEntries` routines.

lefwLayerACFrequency

Writes a `FREQUENCY` statement. The `FREQUENCY` statement is required if you specify a value of 0 in the `lefwLayerACCurrentDensity` routine, and can be used only once in an `ACCURENTDENSITY` statement.

Syntax

```
int lefwLayerACFrequency(
     int numFrequency,
     double* frequency)
```

Arguments

numFrequency

Specifies the number of frequency values.

frequency

Specifies the frequency values, in megahertz.

lefwLayerACTableEntries

Writes a TABLEENTRIES statement. The TABLENTRIES statement is required if you specify a FREQUENCY statement, and can be used only once in an ACCURENTDENSITY statement.

Syntax

```
int lefwLayerACTableEntries(
     int numEntries,
     double* entries)
```

Arguments

numEntries

Specifies the number of table entry values.

entries

Specifies the maximum current for each of the frequency and width pairs specified in the FREQUENCY and WIDTH statements, in milliamps per micron.

lefwLayerACWidth

Writes a WIDTH statement. The WIDTH statement is optional if you specify a FREQUENCY statement, and can be used only once in an ACCURENTDENSITY statement.

Syntax

```
int lefwLayerACWidth(
     int numWidths,
     double* widths)
```

Arguments

numWidths

Specifies the number of width values.

widths

Specifies the wire width values, in microns.

lefwLayerAntennaAreaDiffReducePwl

Writes an ANTENNAAREADIFFREDUCEPWL statement for a routing or cut layer. The ANTENNAAREADIFFREDUCEPWL statement is optional and can be used once after each lefwLayerAntennaModel routine in a LAYER section.

Syntax

```
int lefwLayerAntennaAreaDiffReducePwl(
     int numPwls,
     double* diffAreas,
     double* metalDiffFactors)
```

Arguments

numPwls

Specifies the number of diffusion area and *metalDiffFactor* pairs.

diffAreas

Specifies the *diffArea* values. The values are floating points, specified in microns squared. They should start with 0 and monotonically increase in value to the maximum size *diffArea* expected.

metalDiffFactors

Specifies the *metalDiffFactor* values. The values are floating points with no units and are normally between 0.0 and 1.0.

lefwLayerAntennaAreaFactor

Writes an ANTENNAAREAFACTOR statement. The ANTENNAAREAFACTOR statement is optional and can be used once after each lefwAntennaModel routine in a LAYER section.

Syntax

```
int lefwLayerAntennaAreaFactor(
     double value
     const char* diffUseOnly)
```

Arguments

value

Specifies the adjust or multiply factor for the antenna metal calculation.

diffUseOnly

Optional argument that specifies the current antenna factor should be used only when the corresponding layer is connected to the diffusion. Specify NULL to ignore this argument.

lefwLayerAntennaAreaMinusDiff

Writes an ANTENNAAREAMINUSDIFF statement for a routing or cut layer. The ANTENNAAREAMINUSDIFF statement is optional and can be used once after each lefwLayerAntennaModel routine in a LAYER section.

Syntax

```
int lefwLayerAntennaAreaMinusDiff(
     double minusDiffFactor)
```

Arguments

minusDiffFactor

Specifies the diffusion area. The antenna ratio metal area will subtract the diffusion area connected to it. *minusDiffFactor* is a floating point value and defaults to 0.0.

lefwLayerAntennaAreaRatio

Writes the `ANTENNAAREARATIO` statement. The `ANTENNAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section.

Syntax

```
int lefwLayerAntennaAreaRatio(
    double value)
```

Arguments

value

Specifies the antenna ratio, using the bottom area of the metal wire that is not connected to the diffusion diode.

lefwLayerAntennaCumAreaRatio

Writes an `ANTENNACUMAREARATIO` statement. The `ANTENNACUMAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section.

Syntax

```
int lefwLayerAntennaCumAreaRatio(
    double value)
```

Arguments

value

Specifies the cumulative antenna ratio, using the bottom area of the metal wire that is not connected to the diffusion diode.

lefwLayerAntennaCumDiffAreaRatio

Writes an `ANTENNACUMDIFFAREARATIO` statement. The `ANTENNACUMDIFFAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaCumDiffAreaRatioPWL` in the same `LAYER` section.

Syntax

```
int lefwLayerAntennaCumDiffAreaRatio(
     double value)
```

Arguments

value

Specifies the cumulative antenna ratio, using the bottom area of the metal wire that is connected to the diffusion diode.

lefwLayerAntennaCumDiffAreaRatioPwl

Writes an `ANTENNACUMDIFFAREARATIOPWL` statement. The `ANTENNACUMDIFFAREARATIOPWL` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaCumDiffAreaRatio` in the same `LAYER` section.

Syntax

```
int lefwLayerAntennaCumDiffAreaRatioPwl(
     int numPwls,
     double diffusions,
     double ratios)
```

Arguments

numPwls

Specifies the number of diffusion-ratio pairs.

diffusions

Specifies the diffusion values.

ratios

Specifies the ratio values.

lefwLayerAntennaCumDiffSideAreaRatio

Writes an `ANTENNACUMDIFFSIDEAREARATIO` statement. The `ANTENNACUMDIFFSIDEAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaCumDiffSideAreaRatioPWL` in the same `LAYER` section.

Syntax

```
int lefwLayerAntennaCumDiffSideAreaRatio(
     double value)
```

Arguments

value

Specifies the cumulative antenna ratio, using the side wall area of the metal wire that is connected to the diffusion diode.

lefwLayerAntennaCumDiffSideAreaRatioPwl

Writes an `ANTENNACUMDIFFSIDEAREARATIOPWL` statement. The `ANTENNACUMDIFFSIDEAREARATIOPWL` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaCumDiffSideAreaRatio` in the same `LAYER` section.

Syntax

```
int lefwLayerAntennaCumDiffSideAreaRatioPwl(
     int numPwls,
     double diffusions,
     double ratios)
```

Arguments

numPwls

Specifies the number of diffusion-ratio pairs.

diffusions

Specifies the diffusion values.

ratios

Specifies the ratio values.

lefwLayerAntennaCumSideAreaRatio

Writes an `ANTENNACUMSIDEAREARATIO` statement. The `ANTENNACUMSIDEAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section.

Syntax

```
int lefwAntennaCumSideAreaRatio(
     double value)
```

Arguments

value

Specifies the cumulative antenna ratio, using the side wall area of the metal wire that is not connected to the diffusion diode.

lefwLayerAntennaCumRoutingPlusCut

Writes an `ANTENNACUMROUTINGPLUSCUT` statement for a routing or cut layer. The `ANTENNACUMROUTINGPLUSCUT` statement is optional and can be used once after each `lefwLayerAntennaModel` routine in a `LAYER` section.

Syntax

```
int lefwLayerAntennaCumRoutingPlusCut()
```

lefwLayerAntennaDiffAreaRatio

Writes an `ANTENNADIFFAREARATIO` statement. The `ANTENNADIFFAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaDiffAreaRatioPWL` in the same `LAYER` section.

Syntax

```
int lefwAntennaDiffAreaRatio(
     double value)
```

Arguments

value

Specifies the antenna ratio, using the bottom area of the wire that is connected to the diffusion diode.

lefwLayerAntennaDiffAreaRatioPwl

Writes an `ANTENNADIFFAREARATIOPWL` statement. The `ANTENNADIFFAREARATIOPWL` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaDiffAreaRatio` in the same `LAYER` section.

Syntax

```
int lefwAntennaDiffAreaRatioPWL(
     int numPwls,
     double diffusions,
     double ratios)
```

Arguments

numPwls

Specifies the number of diffusion-ratio pairs.

diffusions

Specifies the diffusion values.

ratios

Specifies the ratio values.

lefwLayerAntennaDiffSideAreaRatio

Writes an `ANTENNADIFFSIDEAREARATIO` statement. The `ANTENNADIFFSIDEAREARATIO` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaDiffSideAreaRatioPwl` in the same `LAYER` section.

Syntax

```
int lefwLayerAntennaDiffSideAreaRatio(
     double value)
```

Arguments

value

Specifies the antenna ratio, using the side wall area of the wire that is connected to the diffusion diode.

lefwLayerAntennaDiffSideAreaRatioPwl

Writes an `ANTENNADIFFSIDEAREARATIOPWL` statement. The `ANTENNADIFFSIDEAREARATIOPWL` statement is optional and can be used once after each `lefwAntennaModel` routine in a `LAYER` section. If you specify this routine, you cannot specify `lefwLayerAntennaDiffSideAreaRatio` in the same `LAYER` section.

Syntax

```
int lefwLayerAntennaDiffSideAreaRatioPwl(
     int numPwls,
     double diffusions,
     double ratios)
```

Arguments

numPwls

Specifies the number of diffusion-ratio pairs.

diffusions

Specifies the diffusion values.

ratios

Specifies the ratio values.

lefwLayerAntennaGatePlusDiff

Writes an `ANTENNAGATEPLUSDIFF` statement for a routing or cut layer. The `ANTENNAGATEPLUSDIFF` statement is optional and can be used once after each `lefwLayerAntennaModel` routine in a `LAYER` section.

Syntax

```
int lefwLayerAntennaGatePlusDiff(
     double plusDiffFactor)
```

Arguments

plusDiffFactor

Specifies that the antenna ratio gate area should include the diffusion area multiplied by *plusDiffFactor*. *minusDiffFactor* is a floating point value.

lefwLayerAntennaModel

Writes an ANTENNAMODEL statement. The ANTENNAMODEL statement is optional and can be used more than once in a LAYER section.

Syntax

```
int lefwLayerAntennaModel(
     const char* oxide)
```

Arguments

oxide

Specifies the oxide model for the layer. Each model can be specified once per layer. If you specify an ANTENNAMODEL statement, that value affects all ANTENNA* statements for the layer that follow it until you specify another ANTENNAMODEL statement.
Value: OXIDE1, OXIDE2, OXIDE3, or OXIDE4

Note: OXIDE1 and OXIDE2 are currently supported. If you specify OXIDE3 or OXIDE4, current tools parse and ignore them.

lefwLayerAntennaSideAreaFactor

Writes an ANTENNASIDEAREAFACTOR statement. The ANTENNASIDEAREAFACTOR statement is optional and can be used once after each lefwAntennaModel routine in a LAYER section.

Syntax

```
int lefwLayerAntennaSideAreaFactor(
     double value
     const char* diffUseOnly)
```

Arguments

value

Specifies the adjust or multiply factor for the antenna metal calculation.

diffUseOnly

Optional argument that specifies that the current antenna factor should only be used when the corresponding layer is connected to the diffusion. Specify NULL to ignore this argument.

lefwLayerAntennaSideAreaRatio

Writes an ANTENNASIDEAREARATIO statement. The ANTENNASIDEAREARATIO statement is optional and can be used once after each lefwAntennaModel routine in a LAYER section.

Syntax

```
int lefwLayerAntennaSideAreaFactor(
     double value)
```

Arguments

value

Specifies the antenna ratio, using the side wall area of the wire that is not connected to the diffusion diode.

lefwLayerDCCurrentDensity

Writes the DCCURRENTDENSITY statement. The DCCURENTDENSITY statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerDCCurrentDensity(
     const char* type,
     double value)
```

Arguments

type

Specifies the DC current limit, AVERAGE.

value

Specifies the current limit for the layer, in milliamps per micron. If you specify 0, you must specify the lefwLayerDCWidth and lefwLayerDCTableEntries routines.

lefwLayerDCTableEntries

Writes a TABLEENTRIES statement. The TABLENTRIES statement is required if you specify a WIDTH statement, and can be used only once in a DCCURENTDENSITY statement.

Syntax

```
int lefwLayerDCTableEntries(
     int numEntries,
     double* entries)
```

Arguments

numEntries

Specifies the number of table entry values.

entries

Specifies the value of current density for each specified width, in milliamps per micron.

lefwLayerDCWidth

Writes a WIDTH statement. The WIDTH statement is required if you specify a value of 0 in the lefwLayerDCCurrentDensity routine, and can be used only once in a DCCURENTDENSITY statement.

Syntax

```
int lefwLayerDCWidth(
     int numWidths,
     double* widths)
```

Arguments

numWidths

Specifies the number of width values.

widths

Specifies the wire width values, in microns.

lefwLayerRouting

Writes the `DIRECTION` and `WIDTH` statements for a `LAYER` section. The `DIRECTION` and `WIDTH` statements are required and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerRouting(
     const char* direction,
     double width)
```

Arguments

direction

Specifies the preferred routing direction.
Value: Specify one of the following:

`HORIZONTAL`	Routing parallel to the x axis is preferred.
`VERTICAL`	Routing parallel to the y axis is preferred.
`DIAG45`	Routing along a 45-degree angle is preferred.
`DIAG135`	Routing along a 135-degree angle is preferred.

width

Specifies the default routing width to use for all regular wiring on the layer.

lefwLayerRoutingArea

Writes an AREA statement. The AREA statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingArea (
     double area)
```

Arguments

area

Specifies the minimum metal area required for polygons on the layer, in distance units squared. All polygons must have an area that is greater than or equal to area, if no MINSIZE rule (lefwLayerRoutingMinsize) is specified. If a MINSIZE rule exists, all polygons must meet either the MINSIZE or the AREA rule.

lefwLayerRoutingCapacitance

Writes a CAPACITANCE CPERSQDIST statement. The CAPACITANCE CPERSQDIST statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingCapacitance(
     const char* capacitance)
```

Arguments

capacitance

Specifies the capacitance for each square unit, in picofarads per square micron.

lefwLayerRoutingCapMultiplier

Writes the CAPMULTIPLIER statement. The CAPMULTIPLIER statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingCapMultiplier(
     double capMultiplier)
```

Arguments

capMultiplier

Specifies the multiplier for interconnect capacitance to account for increases in capacitance caused by nearby wires.

lefwLayerRoutingDiagMinEdgeLength

Writes a `DIAGMINEDGELENGTH` statement. The `DIAGMINEDGELENGTH` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
lefwLayerRoutingDiagMinEdgeLength(
     double diagLength)
```

Arguments

diagLength

Specifies the minimum length for a diagonal edge. Any 45-degree diagonal edge must have a length that is greater than or equal to *diagLength*.

lefwLayerRoutingDiagPitch

Writes a `DIAGPITCH` statement that contains one pitch value that is used for both the 45-degree angle and 135-degree angle directions. The `DIAGPITCH` statement is optional and can only be used once in a `LAYER` section. If you specify this routine, you cannot specify the `lefwLayerRoutingDiagPitchXYDistance` routine.

Syntax

```
lefwLayerRoutingDiagPitch(
     double distance)
```

Arguments

distance

Specifies the 45-degree routing pitch for the layer.

lefwLayerRoutingDiagPitchXYDistance

Writes a `DIAGPITCH` statement that contains separate values for the 45-degree angle and 135-degree angle directions. The `DIAGPITCH` statement is optional and can only be used once in a `LAYER` section. If you specify this routine, you cannot specify the `lefwLayerRoutingDiagPitch` routine.

Syntax

```
lefwLayerRoutingDiagPitchXYDistance(
     double diag45Distance,
     double diag135Distance)
```

Arguments

diag45Distance

Specifies the 45-degree angle pitch (the center-to-center space between 45-degree angle routes).

diag135Distance

Specifies the 135-degree angle pitch.

lefwLayerRoutingDiagSpacing

Writes a `DIAGSPACING` statement. The `DIAGSPACING` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
lefwLayerRoutingDiagSpacing(
     double diagSpacing)
```

Arguments

diagSpacing

Specifies the minimum spacing allowed for a 45-degree angle shape.

lefwLayerRoutingDiagWidth

Writes a DIAGWIDTH statement. The DIAGWIDTH statement is optional and can be used only once in a LAYER section.

Syntax

```
lefwLayerRoutingDiagWidth(
     double diagWidth)
```

Arguments

diagWidth

Specifies the minimum width allowed for a 45-degree angle shape.

lefwLayerRoutingEdgeCap

Writes an EDGECAPACITANCE statement. The EDGECAPACITANCE statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingEdgeCap(
     double edgeCap)
```

Arguments

edgeCap

Specifies a floating-point value of peripheral capacitance, in picoFarads per micron.

lefwLayerRoutingHeight

Writes a HEIGHT statement. The HEIGHT statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingHeight(
     double height)
```

Arguments

height

Specifies the distance from the top of the ground plane to the bottom of the interconnect.

lefwLayerRoutingMaxwidth

Writes a `MAXIMUMWIDTH` statement. The `MAXIMUMWIDTH` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerRoutingMaxwidth(
     double width)
```

Arguments

width

Specifies the maximum width a wire on the layer can have.

lefwLayerRoutingMinenclosedarea

Writes a `MINENCLOSEDAREA` statement. The `MINENCLOSEDAREA` statement is optional and can be used more than once in a `LAYER` section.

Syntax

```
int lefwLayerRoutingMinenclosedarea(
     int numMinenclosed,
     double* area,
     double* width)
```

Arguments

numMinenclosed

Specifies the number of values defined in the routine.

area

Specifies the minimum area size of a hole enclosed by metal. You can specify one or more values.

width

Optional argument that applies the minimum area size limit only when the hole is created from a wire that has a width that is less than or equal to *width*. You can specify one or more values.

lefwLayerRoutingMinimumcut

Writes a MINIMUMCUT statement. The MINIMUMCUT statement is optional and can be used more than once in a LAYER section.

Syntax

```
int lefwLayerRoutingMinimumcut(
     double numCuts,
     double minWidth)
```

Arguments

numCuts

Specifies the number of cuts a via must have when it is on a wide wire or pin whose width is greater than *minWidth*.

minWidth

Specifies the minimum width of the wire or pin.

lefwLayerRoutingMinimumcutConnections

Writes a FROMABOVE or FROMBELOW statement. This statement is optional and can be used only once after each lefwLayerRoutingMinimumcut routine.

Syntax

```
int lefwLayerRoutingMinimumcutConnections(
     const char* direction)
```

Arguments

direction

Specifies the MINIMUMCUT statement applies only to connections from above the layer or from below the layer.
Value: FROMABOVE or FROMBELOW

lefwLayerRoutingMinimumcutLengthWithin

Writes a LENGTH statement. This statement is optional and can be used only once after each lefwLayerRoutingMinimumcut routine.

Syntax

```
int lefwLayerRoutingMinimumcutLengthWithin(
     double length,
     double distance)
```

Arguments

distance

Applies the minimum cut rule to thin wires directly connected to wide wires, if the vias on the thin wires are less than *distance* from the wide wire, and the wide wire has a length that is greater than *length*.

length

Specifies the minimum length of the wide wire.

lefwLayerRoutingMinimumcutWithin

Writes a MINIMUMCUT statement with a WITHIN keyword. This routine is similar to the lefwLayerRoutingMinimumcut routine, except that it lets you specify a WITHIN value. The MINIMUMCUT statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingMinimumcutWithin(
     double numCuts,
     double minWidth,
     double cutDistance)
```

Arguments

numCuts

Specifies the number of cuts a via must have when it is on a wide wire or pin whose width is greater than *minWidth*.

minWidth

Specifies the minimum width of the wire or pin.

cutDistance

Specifies that *numCuts* via cuts must be less than *cutDistance* from each other to be counted together to meet the minimum cut rule.

lefwLayerRoutingMinsize

Writes a `MINSIZE` statement. The `MINSIZE` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
lefwLayerRoutingMinsize(
     int numRect,
     double* minWidth,
     double* minLength)
```

Arguments

numRect

Specifies the number of rectangles defined.

minWidth minLength

Specifies the minimum width and length values for a rectangle that must be able to fit somewhere within each polygon on this layer. All polygons must meet this `MINSIZE` rule, if no `AREA` rule is specified (`lefwLayerRoutingArea`). If an `AREA` rule is specified, all polygons must meet either the `MINSIZE` or the `AREA` rule.

lefwLayerRoutingMinstep

Writes a `MINSTEP` statement. The `MINSTEP` statement is optional and can be used more than once in a `LAYER` section.

Syntax

```
int lefwLayerRoutingMinstep(
     double distance)
```

Arguments

distance
: Specifies the minimum step size, or shortest edge length, for a shape.

lefwLayerRoutingMinstepMaxEdges

Writes a `MINSTEP` statement. This routine is similar to `lefwLayerRoutingMinstep`, except that it lets you specify the `MAXEDGES` option. The `MINSTEP` statement is optional and can be called only once after `lefwStartLayerRouting`.

Syntax

```
int lefwLayerRoutingMinstepMaxEdges(
     double distance,
     double maxEdges)
```

Arguments

distance
: Specifies the minimum step size, or shortest edge length, for a shape.

maxEdges
: Specifies the maximum consecutive edges.

lefwLayerRoutingMinstepWithOptions

Writes a `MINSTEP` statement that contains rule type and total edge length values. The `MINSTEP` statement is optional and can be more than once in a `LAYER` section.

Syntax

```
lefwLayerRoutingMinstepWithOptions(
     double distance,
     const char* rule,
     double maxLength)
```

Arguments

distance

Specifies the minimum step size, or shortest edge length, for a shape.

rule

Indicates to which consecutive edges the MINSTEP rule applies. A DRC violation occurs if one or more consecutive edges of the specified type are less than *distance*. There can only be one rule of each type per layer.
Value: Specify one of the following:

INSIDECORNER	Applies to consecutive edges of an inside corner that are less than *distance*.
OUTSIDECORNER	Applies to consecutive edges of an outside corner that are less than *distance*.
STEP	Applies to consecutive edges of a step that are less than *distance*.

maxLength

Specifies the maximum total edge length allowed that OPC can correct without causing new DRC violations. A violation only occurs if the total length of consecutive edges that are less than *distance* is greater than *maxLength*.

lefwLayerRoutingMinwidth

Writes a MINWIDTH statement. The MINWIDTH statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingMinwidth(
     double width)
```

Arguments

width

Specifies the minimum legal object width on the routing layer, in microns.

lefwLayerRoutingOffset

Writes an `OFFSET` statement that contains one value for both the x and y offsets. The `OFFSET` statement is optional and can be used only once in a `LAYER` section. If you specify this routine, you cannot specify the `lefwLayerRoutingOffsetXYDistance` routine.

Syntax

```
int lefwLayerRoutingOffset(
     double offset)
```

Arguments

offset

Specifies the offset, from the origin (0,0) for the routing grid for the layer.

lefwLayerRoutingOffsetXYDistance

Writes an `OFFSET` statement that contains separate values for the x and y offsets. The `OFFSET` statement is optional and can be used only once in a `LAYER` section. If you specify this routine, you cannot specify the `lefwLayerRoutingOffset` routine.

Syntax

```
lefwLayerRoutingOffsetXYDistance(
     double xDistance,
     double yDistance)
```

Arguments

xDistance

Specifies the x offset for vertical routing tracks.

yDistance

Specifies the y offset for horizontal routing tracks.

lefwLayerRoutingPitch

Writes a `PITCH` statement that contains one pitch value that is used for both the x and y pitch. The `PITCH` statement is required and can be used only once in a `LAYER` section. If you specify this routine, you cannot specify the `lefwLayerRoutingPitchXYDistance` routine.

Syntax

```
int lefwLayerRoutingPitch(
     double pitch)
```

Arguments

pitch

Specifies the routing pitch for the layer.

lefwLayerRoutingPitchXYDistance

Writes a `PITCH` statement that contains separate values for the x and y pitch. The `PITCH` statement is required and can be used only once in a `LAYER` section. If you specify this routine, you cannot specify the `lefwLayerRoutingPitch` routine.

Syntax

```
lefwLayerRoutingPitchXYDistance(
     double xDistance,
     double yDistance)
```

Arguments

xDistance

Specifies the x pitch (that is , the space between each vertical routing track).

yDistance

Specifies the y pitch (that is, the space between each horizontal routing track).

lefwLayerRoutingProtrusion

Writes a `PROTRUSION` statement. The `PROTRUSION` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerRoutingProtrusion(
     double width1,
     double length,
     double width2)
```

Arguments

length

Specifies the maximum length of a protrusion.

width1

Specifies the minimum width of a protrusion.

width2

Specifies the minimum width of the wire to which the protrusion is connected.

lefwLayerRoutingResistance

Writes a `RESISTANCE RPERSQ` statement. The `RESISTANCE RPERSQ` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerRoutingResistance(
     const char* resistance)
```

Arguments

resistance

Specifies the resistance for a square of wire, in ohms per square micron.

lefwLayerRoutingShrinkage

Writes a `SHRINKAGE` statement. The `SHRINKAGE` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerRoutingShrinkage(
     double shrinkage)
```

Arguments

shrinkage

Specifies the value to account for shrinkage of interconnect wiring because of the etching process. Actual wire widths are determined by subtracting this constant value.

lefwLayerRoutingSpacing

Writes a `SPACING` statement. The `SPACING` statement is optional and can be used more than once in a `LAYER` section.

Note: You must use either this routine or the `lefwLayerRoutingStartSpacingtableParallel` routine for all `LAYER` sections.

Syntax

```
int lefwLayerRoutingSpacing(
     double Spacing)
```

Arguments

Spacing

Specifies the minimum spacing allowed between two regular geometries on different nets, also known as the different-net spacing rule.

lefwLayerRoutingSpacingEndOfLine

Writes an `ENDOFLINE` statement. The `ENDOFLINE` statement is optional and can be used only once after a `SPACING` statement.

Syntax

```
int lefwLayerRoutingSpacingEndOfLine(
     double eolWidth,
     double eolWithin)
```

Arguments

eolWidth

Specifies the end-of-line width. An end-of-line with a width less than *eolWidth* requires spacing greater than or equal to *eolSpace* beyond the end of the line anywhere within *eolWIthin* distance.

eolWithin

The *eolWithin* distance. This value must be smaller than the minimum allowed spacing.

lefwLayerRoutingSpacingEOLParallel

Writes a PARALLELEDGE statement. The PARALLELEDGE statement is optional and can be used only once after a SPACING statement.

Syntax

```
int lefwLayerRoutingSpacingEOLParallel(
     double parSpace,
     double parWithin,
     int twoEdges)
```

Arguments

parSpace

Specifies the *parSpace* value. The end-of-line rule applies only if there is a parallel edge less than *parSpace* away that is also less than *parWithin* from the end.

parWithin

Specifies the *parWithin* value.

twoEdges

Optional argument that writes the TWOEDGES keyword, which specifies that the end-of-line rule applies only if there are two parallel edges that meet the PARALLELEDGE *parSpace* and *parWithin* parameters. Specify 0 to ignore this argument.

lefwLayerRoutingSpacingEndOfNotchWidth

Writes an `ENDOFNOTCHWIDTH` statement. The `ENDOFNOTCHWIDTH` statement is optional and can be used only once after a `SPACING` statement.

Syntax

```
int lefwLayerRoutingSpacingEndOfNotchWidth(
     double eonWidth,
     double minNSpacing,
     double minNLength)
```

Arguments

eonWidth

Specifies the end-of-notch width.

minNSpacing

Specifies the minimum notch spacing.

minNLength

Specifies the minimum notch length.

lefwLayerRoutingSpacingLengthThreshold

Writes a `LENGTHTHRESHOLD` statement. The `LENGTHTHRESHOLD` statement is optional and can be used only once after a `lefwLayerRoutingSpacing` routine. If you specify this routine, you cannot specify the `lefwLayerRoutingSpacingRange` or `lefwLayerRoutingSamenet` routines.

Syntax

```
int lefwLayerRoutingSpacingLengthThreshold(
     double lengthValue,
     double minWidth,
     double maxWidth)
```

Arguments

lengthValue

Specifies the maximum parallel run length or projected length with an adjacent metal object.

minWidth, *maxWidth*

Optional arguments that specify a width range. If you specify a range, the threshold spacing rule applies to all objects with widths that are greater than or equal to *minWidth* and less than or equal to *maxWidth*.

lefwLayerRoutingSpacingNotchLength

Writes a `NOTCHLENGTH` statement. The `NOTCHLENGTH` statement is optional and can be used only once after `lefwStartLayerRouting`.

Syntax

```
int lefwLayerRoutingSpacingNotchLength(
     double minNLength)
```

Arguments

minNLength

Specifies the minimum notch length. Any notch with notch length less than *minNLength* must have a notch spacing greater than or equal to the minimum spacing. The value you specify must be only slightly larger than the normal minimum spacing (for example, between 1x or 2x minimum spacing).

lefwLayerRoutingSpacingRange

Writes a `RANGE` statement. The `RANGE` statement is optional and can be used only once after a `lefwLayerRoutingSpacing` routine. If you specify this routine, you cannot specify the `lefwLayerRoutingSpacingLengthThreshold` or `lefwLayerRoutingSameNet` routines.

Syntax

```
int lefwLayerRoutingSpacingRange(
     double minWidth,
     double maxWidth)
```

Arguments

minWidth, *maxWidth*

Specifies a width range. If you specify a range, the minimum spacing rule applies to all wires on the layer with widths that are greater than or equal to *minWidth* and less than or equal to *maxWidth*.

lefwLayerRoutingSpacingRangeInfluence

Writes an `INFLUENCE` statement. The `INFLUENCE` statement is optional and can be used only once after a `lefwLayerRoutingSpacingRange` routine. If you specify this routine, you cannot specify the `lefwLayerRoutingSpacingRangeUseLengthThreshold` or `lefwLayerRoutingSpacingRangeRange` routines in the same `LAYER` section.

Syntax

```
int lefwLayerRoutingSpacingRangeInfluence (
     double infValue,
     double stubMinWidth,
     double stubMaxWidth)
```

Arguments

infValue

Specifies the area of the stub wire which inherits the spacing from a wide wire.

stubMinWidth, stubMaxWidth

Optional arguments that specify a wire width range. If you specify a range, the influence spacing rule applies to all stub wires on the layer with widths that are greater than or equal to *stubMinWidth* and less than or equal to *stubMaxWidth*.

lefwLayerRoutingSpacingRangeRange

Writes a second `RANGE` statement. The second `RANGE` statement is optional and can be used only once after a `lefwLayerRoutingSpacingRange` routine. If you specify this routine, you cannot specify the `lefwLayerRoutingSpacingRangeInfluence` or `lefwLayerRoutingSpacingRangeUseLengthThreshold` routines in the same `LAYER` section.

Syntax

```
int lefwLayerRoutingSpacingRangeRange(
     double minWidth,
     double maxWidth)
```

Arguments

minWidth, *maxWidth*

Specify a second width range. If you specify a second range, the minimum spacing rule applies if the widths of both objects are greater than or equal to *minWidth* and less than or equal to *maxWidth* (each object in a different range).

lefwLayerRoutingSpacingRangeUseLengthThreshold

Writes a USELENGTHTHRESHOLD statement. The USELENGTHTHRESHOLD statement is optional and can be used only once after a lefwLayerRoutingSpacingRange routine. If you specify this routine, you cannot specify the lefwLayerRoutingSpacingRangeRange or lefwLayerRoutingSpacingRangeInfluence routines in the same LAYER section.

This routine is only valid if one or both of the range values in the lefwLayerRoutingSpacingRange routine are not zero.

Syntax

```
int lefwLayerRoutingSpacingRangeUseLengthThreshold()
```

lefwLayerRoutingSpacingSameNet

Writes a SAMENET keyword for a SPACING statement. Only one of lefwLayerRoutingSpacingSameNet, lefwLayerRoutingSpacingRange, or lefwLayerRoutingSpacingLengthThreshold can be called once after lefwLayerRoutingSpacing.

Syntax

```
int lefwLayerRoutingSpacingSameNet(
     int PGOnly)
```

Arguments

PGOnly

Optional argument that specifies the PGONLY keyword. If this keyword is specified, the *minSpacing* value only applies to same-net metal that is a power or ground net.

lefwLayerRoutingStartSpacingtableInfluence

Writes a SPACINGTABLE INFLUENCE statement. The SPACINGTABLE INFLUENCE statement is optional and can be used only once after a lefwLayerRoutingStartSpacingtableParallel routine.

Syntax

```
int lefwLayerRoutingStartSpacingtableInfluence()
```

lefwLayerRoutingStartSpacingInfluenceWidth

Writes a SPACINGTABLE INFLUENCE WIDTH statement. The SPACINGTABLE INFLUENCE WIDTH statement is required if you specify the lefwLayerRoutingStartSpacingtableInfluence routine, and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingStartSpacingInfluenceWidth(
     double width,
     double distance,
     double spacing)
```

Arguments

distance

Specifies an array of values that represent the distance between a wide wire and two perpendicular wires.

spacing

Specifies an array of values that represent the spacing between the two perpendicular wires.

width

Specifies an array of values that represent the width of the wide wire.

lefwLayerRoutingStartSpacingtableParallel

Writes a SPACINGTABLE PARALLELRUNLENGTH statement. The SPACINGTABLE PARALLELRUNLENGTH statement is optional and can be used only once in a LAYER section.

Note: You must use either this routine or the lefwLayerRoutingSpacing routine for all LAYER sections.

Syntax

```
int lefwLayerRoutingStartSpacingtableParallel(
     int numlength,
     double* length)
```

Arguments

length

Specifies an array of values that represent the maximum parallel run length between two wires.

numLength

Specifies the number of *length* values specified.

lefwLayerRoutingStartSpacingtableParallelWidth

Writes a SPACINGTABLE PARALLELRUNLENGTH WIDTH statement. The SPACINGTABLE PARALLELRUNLENGTH WIDTH statement is required if you specify the lefwLayerRoutingStartSpacingtableParallel routine, and can be used only once in a LAYER section.

Syntax

```
int lefwLayerRoutingStartSpacingtableParallelWidth(
     double width,
     int numSpacing,
     double* spacing)
```

Arguments

numSpacing

Specifies the number of *spacing* values specified.

spacing

Specifies an array of values that represent the spacing between the two wires.

width

Specifies and array of values that represent the maximum width of the two wires.

lefwLayerRoutingStartSpacingtableTwoWidths

Writes a `SPACINGTABLE TWOWIDTHS` statement. The `SPACINGTABLE TWOWIDTHS` statement is optional and can be used multiple times in a `LAYER` section after the `lefwLayerRouting` routine.

Syntax

```
int lefwLayerRoutingStartSpacingtableTwoWidths()
```

lefwLayerRoutingStartSpacingtableTwoWidthsWidth

Writes a `SPACINGTABLE TWOWIDTHS WIDTH` statement. This routine is required after a `lefwLayerRoutingStartSpacingtableTwoWidths` routine.

Syntax

```
int lefwLayerRoutingSpacingtableTwoWidthsWidth(
     double width,
     double runLength,
     int numSpacing,
     double* spacing)
```

Arguments

width

The widths of the two objects.

runLength

Optional argument that specifies the parallel run length between the two objects. Specify `0` to ignore this argument.

numSpacing

Specifies the number of *spacing* values provided.

spacing

The spacing values that represent the spacing between two objects.

lefwLayerRoutingEndSpacingtable

Ends a `SPACINGTABLE` statement. This routine is required if you specify `lefwLayerRoutingStartSpacingtableParallel`.

Syntax

```
int lefwLayerRoutineEndSpacingtable()
```

lefwLayerRoutingThickness

Writes a `THICKNESS` statement. The `THICKNESS` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerRoutingThickness(
     double thickness)
```

Arguments

thickness

Specifies the thickness of the interconnect.

lefwLayerRoutingWireExtension

Writes a `WIREEXTENSION` statement. The `WIREEXTENSION` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwLayerRoutingWireExtension(
     double wireExtension)
```

Arguments

wireExtension

Specifies the distance by which wires are extended at vias. Enter `0` to specify no wire extension. Values other than `0` must be more than half of the default routing width for the layer.

lefwMaxAdjacentSlotSpacing

Writes a `MAXADJACENTSLOTSPACING` statement. The `MAXADJACENTSLOTSPACING` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwMaxAdjacentSlotSpacing(
     double maxSpacing)
```

Arguments

maxSpacing

Specifies the maximum spacing, in distance units, allowed between two adjacent slot sections.

lefwMaxCoaxialSlotSpacing

Writes a `MAXCOAXIALSLOTSPACING` statement. The `MAXCOAXIALSLOTSPACING` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwMaxCoaxialSlotSpacing(
     double maxSpacing)
```

Arguments

maxSpacing

Specifies the maximum spacing, in distance units, allowed between two slots in the same slot section.

lefwMaxEdgeSlotSpacing

Writes a MAXEDGESLOTSPACING statement. The MAXEDGESLOTSPACING statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwMaxEdgeSlotSpacing(
     double maxSpacing)
```

Arguments

maxSpacing

Specifies the maximum spacing, in distance units, allowed between slot edges.

lefwMaximumDensity

Writes a MAXIMUMDENSITY statement. The MAXIMUMDENSITY statement is optional and can be used only once in a LAYER section.

Syntax

```
int lefwMaximumDensity(
     double maxDensity)
```

Arguments

maxDensity

Specifies the maximum metal density allowed for the layer, as a percentage of its area.

lefwMinimumDensity

Writes a `MINIMUMDENSITY` statement. The `MINIMUMDENSITY` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwMinimumDensity(
     double minDensity)
```

Arguments

minDensity
Specifies the minimum metal density allowed for the layer, as a percentage of its area.

lefwSlotLength

Writes a `SLOTLENGTH` statement. The `SLOTLENGTH` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwSlotLength(
     double minSlotLength)
```

Arguments

minSlotLength
Specifies the minimum slot length, in distance units, allowed in the design.

lefwSlotWidth

Wries a `SLOTWIDTH` statement. The `SLOTWIDTH` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwSlotWidth(
     double minSlotWidth)
```

Arguments

minSlotWidth

Specifies the minimum slot width, in distance units, allowed in the design.

lefwSlotWireLength

Writes a `SLOTWIRELENGTH` statement. The `SLOTWIRELENGTH` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwSlotWireLength(
     double minWireLength)
```

Arguments

minWireLength

Specifies the minimum wire length, in distance units, allowed for wires that need to be slotted.

lefwSlotWireWidth

Writes a `SLOTWIREWIDTH` statement. The `SLOTWIREWIDTH` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwSlotWireWidth(
     double minWireWidth)
```

Arguments

minWireWidth

Specifies the minimum wire width, in distance units, allowed for wires that need to be slotted.

lefwSplitWireWidth

Writes a `SPLITWIREWIDTH` statement. The `SPLITWIREWIDTH` statement is optional and can be used only once in a `LAYER` section.

Syntax

```
int lefwSplitWireWidth(
     double minWireWidth)
```

Arguments

minWireWidth

Specifies the minimum wire width, in distance units, allowed for wires that need to be split.

Routing Layer Examples

The following example only shows the usage of some functions related to a routing layer. This example is part of the layer callback routine.

```
int layerCB (lefwCallbackType_e type,
                  lefiUserData userData) {
         int    res;
         double *current;

         ...
         res = lefwStartLayerRouting("M3");
         CHECK_RES(res);
         res = lefwLayerRouting("HORIZONTAL", 0.9);
         CHECK_RES(res);
         res = lefwLayerRoutingPitch(1.8);
         CHECK_RES(res);
         res = lefwLayerRoutingWireExtension(8);
         CHECK_RES(res);
         res = lefwLayerRoutingSpacing(0.9, 0, 0);
         CHECK_RES(res);
         res = lefwLayerRoutingResistance("0.0608");
         CHECK_RES(res);
         res = lefwLayerRoutingCapacitance("0.000184");
         CHECK_RES(res);
         res = lefwLayerACCurrentDensity("AVERAGE",0);
         CHECK_RES(res);
         current[0] = 1E6;
         current[1] = 100E6;
         current[2] = 400E6;
```

```
res = lefwLayerACFrequency(3, current);
CHECK_RES(res);
current[0] = 0.6E-6;
current[1] = 0.5E-6;
current[2] = 0.4E-6;
res = lefwLayerACTableEntries(3, current);
CHECK_RES(res);
res = lefwEndLayerRouting("M3");
CHECK_RES(res);
...

return 0;}
```

Macro

Macro routines write a LEF `MACRO` section. A `MACRO` section is optional and can be used more than once in a LEF file. For syntax information about the LEF `MACRO` section, see "Macro" in the *LEF/DEF Language Reference*.

You must begin and end a LEF `MACRO` section with the `lefwStartMacro` and `lefwEndMacro` routines. The *`macroName`* value in the start and end routines identifies the macro being defined. All LEF writer routines that define this macro must be included between the `lefwStartMacro` and `lefwEndMacro` routines specifying that macro name.

For examples of the routines described here, see “Macro Examples” on page 195.

In addition to the routines described in this section, you can include an `OBS`, or `PIN` statement within a `MACRO` section. For more information about these routines, see “Macro Obstruction” on page 195, or “Macro Pin” on page 201.

You can also include a `PROPERTY` statement within a `MACRO` section. For more information about these routines, see “Property” on page 224.

All routines return `0` if successful.

lefwStartMacro

Starts the `MACRO` section. This routine is required to begin each `MACRO` section.

Syntax

```
int lefwStartMacro(
     const char* macroName)
```

Arguments

macroName

Specifies the name of the macro being defined.

lefwEndMacro

Ends the MACRO section for the specified *macroName*.

Syntax

```
int lefwEndMacro(
     const char* macroName)
```

lefwMacroClass

Writes a CLASS statement. The CLASS statement is optional and can be used only once in a MACRO section.

Syntax

```
int lefwMacroClass(
     const char* value1,
     const char* value2)
```

Arguments

value1

Specifies the macro type.
Value: COVER, RING, BLOCK, PAD, CORE, or ENDCAP

value2

Specifies a subtype for a macro type. If *value1* is ENDCAP, you must specify this argument. Otherwise, specify NULL to ignore this argument.

If *Value1* equals:	Then *Value2 is*:
COVER	Optional and can be BUMP.
BLOCK	Optional and can be BLACKBOX or SOFT.

If *Value1* equals:	Then *Value2 is*:
PAD	Optional and can be INPUT, OUTPUT, INOUT, POWER, SPACER, or AREAIO.
CORE	Optional and can be FEEDTHRU, TIEHIGH, TIELOW, SPACER, ANTENNACELL, or WELLTAP.
ENDCAP	Required and can be PRE, POST, TOPLEFT, TOPRIGHT, BOTTOMLEFT, or BOTTOMRIGHT.

lefwMacroEEQ

Writes an EEQ statement. The EEQ statement is optional and can be used only once in a MACRO section.

Syntax

```
int lefwMacroEEQ(
     const char* macroName)
```

Arguments

macroName

Specifies that the macro being defined should be electrically equivalent to the previously defined *macroName*.

lefwMacroForeign

Writes a FOREIGN statement. The FOREIGN statement is optional and can be used more than once in a MACRO section.

Syntax

```
int lefwMacroForeign(
     const char* cellName,
     double xl,
     double yl,
     int orient)
```

Arguments

cellName

Specifies which foreign (GDSII) system name to use when placing an instance of this macro.

xl yl

Optional arguments that specify the macro origin (lower left corner when the macro is in north orientation) offset from the foreign origin. Specify `0` to ignore these arguments.

orient

Optional argument that specifies the orientation of the foreign cell when the macro is in north orientation. Specify `-1` to ignore this argument.
Value: `0` to `7`. For more information, see "Orientation Codes" *on page 21*.

lefwMacroForeignStr

Also writes a `FOREIGN` statement. This routine is the same as the `lefwMacroForeign` routine with the exception of the *orient* argument, which takes a string instead of an integer. The `FOREIGN` statement is optional and can be used more than once in a `MACRO` section.

Syntax

```
int lefwMacroForeignStr(
     const char* cellName,
     double xl,
     double yl,
     const char* orient)
```

Arguments

cellName

Specifies which foreign (GDSII) system name to use when placing an instance of this macro.

xl yl

Optional arguments that specify the macro origin (lower left corner when the macro is in north orientation) offset from the foreign origin. Specify `0` to ignore these arguments.

orient

Optional argument that specifies the orientation of the foreign cell when the macro is in north orientation. Specify `""` to ignore this argument.
Value: `N`, `W`, `S`, `E`, `FN`, `FW`, `FS`, or `FE`

lefwMacroOrigin

Writes an `ORIGIN` statement. The `ORIGIN` statement is optional and can be used only once in a `MACRO` section.

Syntax

```
int lefwMacroOrigin(
     double xl,
     double yl)
```

Arguments

xl, *yl*

Specifies the origin of the macro. *xl*, *yl* is the lower left corner point of the macro. The coordinates for macro sites, ports, and obstructions are specified with respect to the macro origin. The origin itself is specified with respect to the lower left corner of the bounding box of the sites of the macro.

lefwMacroSite

Writes a `SITE` statement. The `SITE` statement is optional and can be used more than once in a `MACRO` section.

Syntax

```
int lefwMacroSite(
     const char* siteName)
```

Arguments

siteName

Specifies the site associated with the macro.

lefwMacroSitePattern

Writes a `SITE` statement that includes a site pattern. The site pattern indicates that the cell is a gate-array cell rather that a row-based standard cell. The `SITE` statement is optional and can be used more than once in a `MACRO` section.

Syntax

```
lefwMacroSitePattern(
     const char* name,
     double origX,
     double orgY,
     int orient,
     int numX,
     int numY,
     double spaceX,
     double spaceY)
```

Arguments

name

Specifies the site associated with the macor.

origX origY

Optional arguments that specify the origin of the site inside the macro. Specify `0` to ignore these arguments.

orient

Optional argument that specifies the orientation of the site at that location. Specify `-1` to ignore this argument.
Value: `0` to `7`. For more information, see "Orientation Codes" *on page 21*.

numX numY

Optional arguments that specify the number of sites to add in the x and y directions. Specify `0` to ignore these arguments.

spaceX spaceY

Optional arguments that specify the spacing between sites in the x and y directions. Specify `0` to ignore these arguments.

lefwMacroSitePatternStr

Also writes a `SITE` statement that includes a site pattern. This routine is the same as the `lefwMacroSitePattern` routine with the exception of the *`orient`* argument, which takes a string instead of an integer. The `SITE` statement is optional and can be used more than once in a `MACRO` section.

Syntax

```
lefwMacroSitePatternStr(
     const char* name,
     double origX,
     double orgY,
     int orient,
     int numX,
     int numY,
     double spaceX,
     double spaceY)
```

Arguments

`name`

Specifies the site associated with the macor.

`origX origY`

Optional arguments that specify the origin of the site inside the macro. Specify `0` to ignore these arguments.

`orient`

Optional argument that specifies the orientation of the site at that location. Specify `""` to ignore this argument.
Value: `N`, `W`, `S`, `E`, `FN`, `FW`, `FS`, or `FE`

`numX numY`

Optional arguments that specify the number of sites to add in the x and y directions. Specify `0` to ignore these arguments.

`spaceX spaceY`

Optional arguments that specify the spacing between sites in the x and y directions. Specify `0` to ignore these arguments.

lefwMacroSize

Writes a `SIZE` statement. The `SIZE` statement is required and can be used only once in a `MACRO` section.

Syntax

```
int lefwMacroSize(
     double width,
     double height)
```

Arguments

width, height

Specify the minimum bounding rectangle, in microns, for the macro. The bounding rectangle should be a multiple of the placement grid.

lefwMacroSymmetry

Writes a `SYMMETRY` statement. The `SYMMETRY` statement is optional and can be used only once in a `MACRO` section.

Syntax

```
int lefwMacroSymmetry(
     const char* symmetry)
```

Arguments

symmetry

Specifies the allowable orientations for the macro.
Value: `X`, `Y`, or `R90`

lefwStartMacroDensity

Starts a `DENSITY` statement in the `MACRO` statement.The `DENSITY` statement is optional and can be used only once in a `MACRO` statement.

Each `DENSITY` statement must start with this routine and end with the `lefwEndMacroDensity` routine. Each `DENSITY` statement also must include at least one `lefwMacroDensityLayerRect` routine.

Syntax

```
lefwStartMacroDensity(
     const char* layerName)
```

Arguments

layerName

Specifies the layer on which to create the density rectangles.

lefwMacroDensityLayerRect

Writes a `RECT` statement in the `DENSITY` statement. The `RECT` statement is required and can be used more than once in a `DENSITY` statement.

Syntax

```
lefwMacroDensityLayerRect(
     double x1,
     double y1,
     double x2,
     double y2,
     double densityValue)
```

Arguments

x1 y1 x2 y2

Specifies the coordinates of a rectangle.

densityValue

Specifies the percentage density of the rectangle.
Value: `0` to `100`

lefwEndMacroDensity

Ends the `DENSITY` statement.

Syntax

```
lefwEndMacroDensity()
```

Macro Examples

The following example shows a callback routine with the type `lefwMacroCbkType`. This example shows function calls to create a macro. It does not include function calls to create a macro obstruction. For an example of how to create a macro obstruction, see the Macro Obstruction section. This example only shows the usage of some functions related to Macro.

```
int macroCB (lefwCallbackType_e type,
                  lefiUserData userData) {
          int    res;
          double *xpath;
          double *ypath;

          // Check if the type is correct
          if (type != lefwMacroCbkType) {
              printf("Type is not lefwMacroCbkType, terminate
                writing.\n");
              return 1;
          }

          res = lefwStartMacro("INV");
          CHECK_RES(res);
          res = lefwMacroClass("CORE", NULL);
          CHECK_RES(res);
          res = lefwMacroForeign("INVS", 0, 0, -1);
          CHECK_RES(res);
          res = lefwMacroPower(1.0);
          CHECK_RES(res);
          res = lefwMacroSize(67.2, 24);
          CHECK_RES(res);
          res = lefwMacroSymmetry("X Y R90");
          CHECK_RES(res);
          res = lefwMacroSite("CORE1");
          CHECK_RES(res);

          return 0;}
```

Macro Obstruction

Macro obstruction routines write an `OBS` (macro obstruction) section, which further defines a macro. An `OBS` section is optional and can be used more than once in a `MACRO` section. For syntax information about the LEF `OBS` section, see "Macro Obstruction Statement" in the *LEF/DEF Language Reference*.

You must use the `lefwStartMacroObs` and `lefwEndMacroObs` routines to start and end the `OBS` section. The remaining macro obstruction routines described in this section must be included between these routines.

For examples of the routines described here, see "Macro Obstruction Examples" on page 201.

All routines return `0` if successful.

lefwStartMacroObs

Starts the `OBS` section in a `MACRO` section. This routine is required for each `OBS` section, and can be used more than once in a `MACRO` section.

Syntax

```
int lefwStartMacroObs()
```

lefwEndMacroObs

Ends the `OBS` section.

Syntax

```
int lefwEndMacroObs()
```

lefwMacroObsDesignRuleWidth

Writes a `DESIGNRULEWIDTH` statement. Either a `LAYER` statement, a `DESIGNRULEWIDTH` statement, or a `VIA` statement must be defined within an `OBS` section and can be used more than once.

Syntax

```
int lefwMacroObsDesignRuleWidth(
     const char* layerName
     double width)
```

Arguments

layerName

Specifies the layer on which the geometry lies.

width

Optional argument that specifies the effective design rule width. If specified, the obstruction is treated as a shape of this width for all spacing checks. Specify `0` to ignore this argument.

lefwMacroObsLayer

Writes a `LAYER` statement. Either a `LAYER` statement, a `DESIGNRULEWIDTH` statement, or a `VIA` statement must be defined within an `OBS` section and can be used more than once.

Syntax

```
int lefwMacroObsLayer(
     const char* layerName,
     double spacing)
```

Arguments

layerName

Specifies the layer on which to place the obstruction.

spacing

Optional argument that specifies the minimum spacing allowed between this obstruction and any other shape. Specify `0` to ignore this argument.

lefwMacroObsLayerPath

Writes a `PATH` statement. Either a `PATH`, `POLYGON`, or `RECT` statement must follow a `LAYER` statement and can be used more than once.

Syntax

```
int lefwMacroObsLayerPath(
     int num_paths,
     double* x1,
     double* y1,
```

```
     int numX,
     int numY,
     double spaceX,
     double spaceY)
```

Arguments

numPaths

Specifies the number of paths to create.

x1 y1

Creates a path between the specified points. The path automatically extends the length by half of the current width on both end points to form a rectangle. (A previous `WIDTH` statement is required.) The line between each pair of points must be parallel to the x or y axis (45-degree angles are not allowed).

numX numY spaceX spaceY

Optional arguments that specify the `PATH ITERATE` statement. *numX* and *numy* specify the number of columns and rows of points that make up the array. *spaceX* and spaceY specify the spacing, in distance units, between the columns and rows. Specify `0` to ignore these arguments.

lefwMacroObsLayerPolygon

Writes a `POLYGON` statement. Either a `PATH`, `POLYGON`, or `RECT` statement must follow a `LAYER` statement and can be used more than once.

Syntax

```
int lefwMacroObsLayerPolygon(
     int num_polys,
     double* x1,
     double* y1,
     int numX,
     int numY,
     double spaceX,)
     double spaceY)
```

Arguments

num_polys

Specifies the number of polygon sides.

x1 y1

Specifies a sequence of points to generate a polygon geometry. Every polygon edge must be parallel to the x or y axis, or at a 45-degree angle.

numX numY spaceX spaceY

Optional arguments that specify the `POLYGON ITERATE` statement. *numX* and *numy* specify the number of columns and rows of points that make up the array. *spaceX* and spaceY specify the spacing, in distance units, between the columns and rows. Specify `0` to ignore these arguments.

lefwMacroObsLayerRect

Writes a `RECT` statement. Either a `PATH`, `POLYGON`, or `RECT` statement must follow a `LAYER` statement and can be used more than once.

Syntax

```
int lefwMacroObsLayerRect(
     double xl1,
     double yl1,
     double xl2,
     double yl2,
     int numX,
     int numY,)
     double spaceX,
     double spaceY)
```

Arguments

xl1 yl1 xl2 yl2

Specifies a rectangle in the current layer, where the points specified are opposite corners of the rectangle.

numX numY spaceX spaceY

Optional arguments that specify the `RECT ITERATE` statement. *numX* and *numy* specify the number of columns and rows of points that make up the array. *spaceX* and spaceY specify the spacing, in distance units, between the columns and rows. Specify `0` to ignore these arguments.

lefwMacroObsLayerWidth

Writes a `WIDTH` statement. The `WIDTH` statement is optional and can be used only once in an `LAYER` section.

Syntax

```
int lefwMacroObsLayerWidth(
     double width)
```

Arguments

width

Specifies the width that the `PATH` statements use.

lefwMacroObsVia

Writes a `VIA` statement. Either a `LAYER` statement, a `DESIGNRULEWIDTH` statement, or a `VIA` statement must be defined within an `OBS` section and can be used more than once.

Syntax

```
int lefwMacroObsVia(
     double x1,
     double y1,
     const char* viaName,
     int numX,
     int numY,
     double spaceX,)
     double spaceY)
```

Arguments

x1 y1

Specify the location to place the via.

viaName

Specifies the name of the via to place.

numX numY spaceX spaceY

Optional arguments that specify the `VIA ITERATE` statement. *numX* and *numy* specify the number of columns and rows of points that make up the array. *spaceX* and spaceY specify the spacing, in distance units, between the columns and rows. Specify `0` to ignore these arguments.

Macro Obstruction Examples

The following example only shows the usage of some functions related to Macro Obstruction. This example is part of the Macro callback routine.

```
int macroCB (lefwCallbackType_e type,
                  lefiUserData userData) {
         int    res;
         double *xpath;
         double *ypath;

         ...
         res = lefwStartMacroObs();
         CHECK_RES(res);
         res = lefwMacroObsLayer("M1", 0);
         CHECK_RES(res);
         res = lefwMacroObsLayerRect(24.1, 1.5, 43.5, 208.5, 0,
          0, 0, 0);
         CHECK_RES(res);
         xpath = (double*)malloc(sizeof(double)*2);
         ypath = (double*)malloc(sizeof(double)*2);
         xpath[0] = 8.4;
         ypath[0] = 3;
         xpath[1] = 8.4;
         ypath[1] = 124;
         res = lefwMacroObsLayerPath(2, xpath, ypath, 0, 0, 0, 0);
         CHECK_RES(res);
         free((char*)xpath);
         free((char*)ypath);
         res = lefwEndMacroObs();
         CHECK_RES(res);
         ...

         return 0;}
```

Macro Pin

Macro Pin routines write a `PIN` section, which further defines a macro. A `PIN` section is optional in each `MACRO` section and can be defined more than once in a `MACRO` section. For syntax information about the LEF PIN section, see "Macro Pin Statement" in the *LEF/DEF Language Reference*.

You must use the lefwStartMacroPin and lefwEndMacroPin routines to start and end the PIN section. The remaining macro pin routines must be included between these routines.

For examples of the routines described here, see "Macro Pin Examples" on page 211.

In addition to the routines described in this section, you can include a PORT section within a PIN section. For more information about these routines, see "Macro Pin Port" on page 211.

All routines return 0 if successful.

lefwStartMacroPin

Starts the PIN section in a MACRO section. This routine is required for each PIN section and can be used more than once in a MACRO section.

Syntax

```
int lefwStartMacroPin(
     const char* pinName)
```

Arguments

pinName

Specifies the name of the library pin.

lefwEndMacroPin

Ends the PIN section for the specified pin.

Syntax

```
int lefwEndMacroPin(
     const char* pinName)
```

Arguments

pinName

Specifies the name of the library pin.

lefwMacroPinAntennaDiffArea

Writes an ANTENNADIFFAREA statement. The ANTENNADIFFAREA statement is optional and can be used more than once in a PIN section.

Syntax

```
int lefwMacroPinAntennaDiffArea(
     double value,
     const char* layerName)
```

Arguments

value

Specifies the diffusion area, in micron-squared units, to which the pin is connected on a layer.

layerName

Optional argument that specifies the layer. If you do not specify a layer name, *value* applies to all layers. Specify NULL to ignore this argument.

lefwMacroPinAntennaGateArea

Writes an ANTENNAGATEAREA statement. The ANTENNAGATEAREA statement is optional and can be used once after each lefwMacroPinAntennaModel routine in a PIN section.

Syntax

```
int lefwMacroPinAntennaGateArea(
     double value,
     const char* layerName)
```

Arguments

value

Specifies the gate area, in micron-squared units, to which the pin is connected on a layer.

layerName

Optional argument that specifies the layer. If you do not specify a layer name, *value* applies to all layers. Specify NULL to ignore this argument.

lefwMacroPinAntennaMaxAreaCar

Writes an `ANTENNAMAXAREACAR` statement. The `ANTENNAMAXAREACAR` statement is optional and can be used once after each `lefwMacroPinAntennaModel` routine in a `PIN` section.

Syntax

```
int lefwMacroPinAntennaMaxAreaCar(
     double value,
     const char* layerName)
```

Arguments

value

For hierarchical process antenna effect calculation, specifies the maximum cumulative antenna ratio value on the specified *layerName*, using the cut area below the current pin layer.

layerName

Specifies the layer.

lefwMacroPinAntennaMaxCutCar

Writes an `ANTENNAMAXCUTCAR` statement. The `ANTENNAMAXCUTCAR` statement is optional and can be used once after each `lefwMacroPinAntennaModel` routine in a `PIN` section.

Syntax

```
int lefwMacroPinAntennaMaxCutCar(
     double value,
     const char* layerName)
```

Arguments

value

For hierarchical process antenna effect calculation, specifies the maximum cumulative antenna ratio value on the specified *layerName*, using the cut area below the current pin layer.

layerName

Specifies the layer.

lefwMacroPinAntennaMaxSideAreaCar

Writes an ANTENNAMAXSIDEAREACAR statement. The ANTENNAMAXSIDEAREACAR statement is optional and can be used once after each lefwMacroPinAntennaModel routine in a PIN section.

Syntax

```
int lefwMacroPinAntennaMaxSideAreaCar(
     double value,
     const char* layerName)
```

Arguments

value

For hierarchical process antenna effect calculation, specifies the maximum cumulative antenna ratio value on the specified *layerName*, using the metal side wall area below the current pin layer.

layerName

Specifies the layer.

lefwMacroPinAntennaModel

Writes an ANTENNAMODEL statement. The ANTENNAMODEL statement is optional and can be used more than once in a PIN section.

Syntax

```
int lefwMacroPinAntennaModel(
     const char* oxide)
```

Arguments

oxide

Specifies the oxide model for the pin. Each model can be specified once per layer. If you specify an ANTENNAMODEL statement, that value affects all ANTENNAGATEAREA and

ANTENNA*CAR statements for the pin that follow it until you specify another ANTENNAMODEL statement.
Value: OXIDE1, OXIDE2, OXIDE3, or OXIDE4

Note: OXIDE1 and OXIDE2 are currently supported. If you specify OXIDE3 or OXIDE4, current tools parse and ignore them.

lefwMacroPinAntennaPartialCutArea

Writes an ANTENNAPARTIALCUTAREA statement. The ANTENNAPARTIALCUTAREA statement is optional and can be used more than once in a PIN section.

Syntax

```
int lefwMacroPinAntennaPartialCutArea(
     double value,
     const char* layerName)
```

Arguments

value

Specifies the partial cut area, which is above the current pin layer and inside, or outside, the macro on a layer.

layerName

Optional argument that specifies the layer. If you specify a layer name, *value* applies to antennas on that layer only. If you do not specify a layer name, *value* applies to all layers. Specify NULL to ignore this argument.

lefwMacroPinAntennaPartialMetalArea

Writes an ANTENNAPARTIALMETALAREA statement. The ANTENNAPARTIALMETALAREA statement is optional and can be used more than once in a PIN section.

Syntax

```
int lefwMacroPinAntennaPartialMetalArea(
     double value,
     const char* layerName)
```

Argument

value

Specifies the partial metal area, which is connected directly to the I/O pin and the inside, or outside, of the macro on a layer.

layerName

Optional argument that specifies the layer. If you do not specify a layer name, *value* applies to all layers. Specify NULL to ignore this argument.

lefwMacroPinAntennaPartialMetalSideArea

Writes an ANTENNAPARTIALMETALSIDEAREA statement. The ANTENNAPARTIALMETALSIDEAREA statement is optional and can be used more than once in a PIN section.

Syntax

```
int lefwMacroPinAntennaPartialMetalSideArea(
     double value,
     const char* layerName)
```

Arguments

value

Specifies the partial metal side wall area, which is connected directly to the I/O pin and inside, or outside, of the macro on a layer.

layerName

Optional argument that specifies the layer. If you do not specify a layer name, *value* applies to all layers. Specify NULL to ignore this argument.

lefwMacroPinDirection

Writes a DIRECTION statement. The DIRECTION statement is optional and can be used only once in a PIN section.

Syntax

```
int lefwMacroPinDirection(
     const char* direction)
```

Arguments

direction

Specifies the pin type.
Value: INPUT, OUTPUT, OUTPUT TRISTATE, INOUT, or FEEDTHRU

lefwMacroPinGroundSensitivity

Writes a GROUNDSENSITIVITY statement. The GROUNDSENSITIVITY statement is optional and can be used only once in a PIN section.

Syntax

```
lefwMacroPinGroundSensitivity(
     const char* pinName)
```

Arguments

pinName

Specifies that if this pin is connected to a tie-low connection (such as 1'b0 in Verilog), it should connect to the same net to which *pinName* is connected.

lefwMacroPinMustjoin

Writes a MUSTJOIN statement. The MUSTJOIN statement is optional and can be used only once in a PIN section.

Syntax

```
int lefwMacroPinMustjoin(
     const char* pinName)
```

Arguments

pinName

Specifies the name of another pin in the cell that must be connected with the pin being defined.

lefwMacroPinNetExpr

Wries a NETEXPR statement in a PIN section. The NETEXPR statement is optional and can be used only once in a PIN section.

Syntax

```
lefwMacroPinNetExpr(
     const char* name)
```

Arguments

name

Specifies a net expression property name (such as power1 or power2). If *name* matches a net expression property in the netlist (such as in Verilog, VHDL, or OpenAccess), then the property is evaluated, and the software identifies a net to which to connect this pin.

lefwMacroPinShape

Writes a SHAPE statement. The SHAPE statement is optional and can be used only once in a PIN section.

Syntax

```
int lefwMacroPinShape(
     const char* name)
```

Arguments

name

Specifies a pin with special connection requirements because of its shape.
Value: ABUTMENT, RING, or FEEDTHRU

lefwMacroPinSupplySensitivity

Writes a SUPPLYSENSITIVITY statement. The SUPPLYSENSITIVITY statement is optional and can be used only once in a PIN section.

Syntax

```
lefwMacroPinSupplySensitivity(
     const char* pinName)
```

Arguments

pinName

Specifies that if this pin is connected to a tie-high connection (such as `1'b1` in Verilog), it should connect to the same net to which *pinName* is connected.

lefwMacroPinTaperRule

Writes a `TAPERRULE` statement. The `TAPERRULE` statement is optional and can be used only once in a `PIN` section.

Syntax

```
int lefwMacroPinTaperRule(
     const char* ruleName)
```

Arguments

ruleName

Specifies the nondefault rule to use when tapering wires to the pin.

lefwMacroPinUse

Writes a `USE` statement. The `USE` statement is optional and can be used only once in a `PIN` section.

Syntax

```
int lefwMacroPinUse(
     const char* use)
```

Arguments

use

Specifies how the pin is used.
Value: SIGNAL, ANALOG, POWER, GROUND, or CLOCK

Macro Pin Examples

The following example only shows the usage of some functions related to Macro Pin.This example is part of the Macro callback routine.

```
int macroCB (lefwCallbackType_e type,
                  lefiUserData userData) {
         int    res;

         ...
         res = lefwStartMacroPin("Z");
         CHECK_RES(res);
         res = lefwMacroPinDirection("OUTPUT");
         CHECK_RES(res);
         res = lefwMacroPinUse("SIGNAL");
         CHECK_RES(res);
         res = lefwMacroPinShape("ABUTMENT");
         CHECK_RES(res);
         res = lefwMacroPinPower(0.1);
         CHECK_RES(res);
         res = lefwStartMacroPinPort(NULL);
         CHECK_RES(res);
         res = lefwEndMacroPin("Z");
         CHECK_RES(res);
         ...

         return 0;}
```

Macro Pin Port

Macro Pin Port routines write a PORT section, which further defines a macro pin. The PORT section is required for each PIN section and can be used more than once in a PIN section. For syntax information about the LEF PIN section, see "Macro Pin Statemen"t in the *LEF/DEF Language Reference*.

You must use the lefwStartMacroPinPort and lefwEndMacroPinPort routines to start and end the PORT section. The lefwStartMacroPinPort routine must be called after the lefwStartMacroPin routine.The remaining port routines must be included between these routines.

For examples of the routines described here, see "Macro Pin Port Examples" on page 217.

All routines return `0` if successful.

lefwStartMacroPinPort

Starts the `PORT` section.

Syntax

```
int lefwStartMacroPinPort(
     const char* classType)
```

Arguments

classType

Optional argument that specifies whether or not the port is a core port.
Value: `NONE` or `CORE`.

lefwEndMacroPinPort

Ends the `PORT` section.

Syntax

```
int lefwEndMacroPinPort()
```

lefwMacroPinPortDesignRuleWidth

Writes a `DESIGNRULEWIDTH` statement. Either a `LAYER` statement, a `DESIGNRULEWIDTH` statement, or a `VIA` statement must be defined in a `PORT` section and can be used more than once.

Syntax

```
int lefwMacroPinPortDesignRuleWidth(
     const char* layerName,
     double width)
```

Argument

layerName

Specifies the layer on which to place the geometry.

width

Optional argument that specifies the effective design rule width. If specified, the router uses the spacing defined in the layer section that corresponds to *width*. Specify 0 to ignore this argument.

lefwMacroPinPortLayer

Writes a LAYER statement in the PORT section. Either a LAYER statement, a DESIGNRULEWIDTH statement, or a VIA statement must be defined in a PORT section and can be used more than once.

Syntax

```
int lefwMacroPinPortLayer(
     const char* layerName,
     double spacing)
```

Arguments

layerName

Specifies the layer on which to place the geometry.

spacing

Optional argument that specifies the minimum spacing allowed between this geometry and any other shape. Specify 0 to ignore this argument.

lefwMacroPinPortLayerPath

Writes a PATH statement. Either a PATH, POLYGON, or RECT statement must follow a LAYER statement.

Syntax

```
int lefwMacroPinPortLayerPath(
     int num_paths,
     double* xl,
```

```
     double* y1,
     int numX,
     int numY,
     double spaceX,
     double spaceY)
```

Arguments

numPaths

Specifies the number of paths to create.

x1 y1

Create a path between the specified points. The path automatically extends the length by half of the current width on both end points to form a rectangle. (A previous `WIDTH` statement is required.) The line between each pair of points must be parallel to the x or y axis (45-degree angles are not allowed).

numX numY spaceX spaceY

Optional arguments that specify the `PATH ITERATE` statement. *numX* and *numy* specify the number of columns and rows of points that make up the array. *spaceX* and spaceY specify the spacing, in distance units, between the columns and rows. Specify `0` to ignore these arguments.

lefwMacroPinPortLayerPolygon

Writes a `POLYGON` statement. Either a `PATH`, `POLYGON`, or `RECT` statement must follow a `LAYER` statement.

Syntax

```
int lefwMacroPinPortLayerPolygon(
     int num_polys,
     double* x1,
     double* y1,
     int numX,
     int numY,
     double spaceX,
     double spaceY)
```

Arguments

num_polys

Specifies the number of polygon sides.

x1 y1

Specifies a sequence of points to generate a polygon geometry. Each polygon edge must be parallel to the x or y axis, or at a 45-degree angle.

numX numY spaceX spaceY

Optional arguments that specify the `POLYGON ITERATE` statement. *numX* and *numy* specify the number of columns and rows of points that make up the array. *spaceX* and spaceY specify the spacing, in distance units, between the columns and rows. Specify `0` to ignore these arguments.

lefwMacroPinPortLayerRect

Writes a `RECT` statement. Either a `PATH`, `POLYGON`, or `RECT` statement must follow a `LAYER` statement.

Syntax

```
int lefwMacroPinPortLayerRect(
     double xl1,
     double yl1,
     double xl2,
     double yl2,
     int numX,
     int numY,
     double spaceX,
     double spaceY)
```

Arguments

xl1 yl1 xl2 yl2

Specifies a rectangle in the current layer, where the points specified are opposite corners of the rectangle.

numX numY spaceX spaceY

Optional arguments that specify the `RECT ITERATE` statement. *numX* and *numy* specify the number of columns and rows of points that make up the array. *spaceX* and spaceY specify the spacing, in distance units, between the columns and rows. Specify `0` to ignore these arguments.

lefwMacroPinPortLayerWidth

Writes a `WIDTH` statement. The `WIDTH` statement is optional and can be used only once in a `PORT` section.

Syntax

```
int lefwMacroPinPortLayerWidth(
     double width)
```

Arguments

width

Specifies the width that the `PATH` statements use.

lefwMacroPinPortVia

Writes a `VIA` statement. Either a `LAYER` statement, a `DESIGNRULEWIDTH` statement, or a `VIA` statement must be defined in a `PORT` section and can be used more than once.

Syntax

```
int lefwMacroPinPortVia(
     double x1,
     double y1,
     const char* viaName,
     int numX,
     int numY,
     double spaceX,
     double spaceY)
```

Arguments

x1 y1

Specify the location to place the via.

viaName

Specifies the name of the via to place.

numX numY spaceX spaceY

Optional arguments that specify the `VIA ITERATE` statement. *numX* and *numy* specify the number of columns and rows of points that make up the array. *spaceX* and spaceY specify the spacing, in distance units, between the columns and rows. Specify `0` to ignore these arguments.

Macro Pin Port Examples

The following example only shows the usage of some functions related to Macro Pin Port. This example is part of the Macro callback routine.

```
int macroCB (lefwCallbackType_e type,
                  lefiUserData userData) {
         int    res;
         double *xpath;
         double *ypath;

         ...
         res = lefwStartMacroPin("Z");
         CHECK_RES(res);

         ...

         res = lefwStartMacroPinPort(NULL);
         CHECK_RES(res);
         res = lefwMacroPinPortLayer("M2", 5.6);
         CHECK_RES(res);
         xpath = (double*)malloc(sizeof(double)*3);
         ypath = (double*)malloc(sizeof(double)*3);
         xpath[0] = 30.8;
         ypath[0] = 9;
         xpath[1] = 42;
         ypath[1] = 9;
         xpath[2] = 30.8;
         ypath[2] = 9;
         res = lefwMacroPinPortLayerPath(3, xpath, ypath, 0, 0,
          0, 0);
         CHECK_RES(res);
         res = lefwEndMacroPinPort();
         CHECK_RES(res);

         ...
         res = lefwEndMacroPin("Z");
         CHECK_RES(res);
         free((char*)xpath);
         free((char*)ypath);
```

```
...

return 0;}
```

Manufacturing Grid

The Manufacturing Grid routine writes a LEF `MANUFACTURINGGRID` statement. The `MANUFACTURINGGRID` statement is optional and can be used only once in a LEF file. For syntax information about the `MANUFACTURINGGRID` statement, see "Manufacturing Grid" in the *LEF/DEF Language Reference*.

This routine returns 0 if successful.

lefwManufacturingGrid

Writes a `MANUFACTURINGGRID` statement.

Syntax

```
int lefwManufacturingGrid(
     double grid)
```

Arguments

grid

Specifies the value for the manufacturing grid. You must specify a positive number for a value.

Maximum Via Stack

The Maximum Stack Via routine writes a LEF `MAXVIASTACK` statement. The `MAXVIASTACK` statement is optional and can be used only once in a LEF file. For syntax information about the `MAXVIASTACK` statement, see "Maximum Via Stack" in the *LEF/DEF Language Reference*.

Important

The `lefwMaxviastack` routine must be used only after all layer routines are used.

This routine returns `0` if successful.

lefwMaxviastack

Writes a `MAXVIASTACK` statement.

Syntax

```
int lefwMaxviastack(
     int value,
     const char* bottomlayer,
     const char* topLayer)
```

Arguments

value
: Specifies the maximum allowed number of single-stacked vias.

bottomLayer
: Optional argument that specifies the bottom layer in a range of layers for which the maximum stacked via rule applies. Specify `NULL` to ignore this argument.

topLayer
: Optional argument that specifies the top layer in a range of layers for which the maximum stacked via rule applies. Specify `NULL` to ignore this argument.

Nondefault Rule

Nondefault Rule routines write a LEF `NONDEFAULTRULE` statement. The `NONDEFAULTRULE` statement is optional and can be used only once in a LEF file. For syntax information about the LEF `NONDEFAULTRULE` statement, see "Nondefault Rule" in the *LEF/DEF Language Reference*.

You must use the `lefwStartNondefaultRules` and `lefwEndNondefaultRules` routines to start and end the `NONDEFAULTRULE` section. The `lefwNonDefaultRuleLayer` routine must be included between these routines.

For examples of the routines described here, see "Nondefault Rules Example" on page 224.

In addition to the routines described in this section, you can include a `PROPERTY` statement and a `VIA` statement within a `NONDEFAULTRULE` section. For more information about these routines, see "Property" on page 224, or "Via" on page 240.

All routines return `0` if successful.

lefwStartNonDefaultRule

Starts the `NONDEFAULTRULE` statement.

Syntax

```
int lefwStartNonDefaultRule(
     const char* ruleName)
```

Arguments

ruleName
: Specifies the name of the nondefault rule to define.

lefwEndNonDefaultRule

Ends the `NONDEFAULTRULE` statement for the specified *ruleName*.

Syntax

```
int lefwEndNonDefaultRule(
     const char* ruleName)
```

lefwNonDefaultRuleHardspacing

Writes a `HARDSPACING` statement. The `HARDSPACING` statement specifies that any spacing values that exceed the LEF `LAYER` spacing requirements are "hard" rules instead of "soft" rules. By default, routers treat extra spacing requirements as soft rules that are high cost to violate, but not real spacing violations. The `HARDSPACING` statement is optional and can be used only once in a `NONDEFAULTRULE` statement.

Syntax

```
lefwNonDefaultRuleHardspacing()
```

lefwNonDefaultRuleLayer

Writes a `LAYER` statement in the `NONDEFAULTRULE` statement. The `LAYER` statement is required and can be used more than once in a `NONDEFAULTRULE` statement.

Syntax

```
int lefwNonDefaultRuleLayer(
     const char* layerName,
     double width,
     double minSpacing,
     double wireExtension,
     double resistance,
     double capacitance,
     double edgecap)
```

Arguments

layerName

Specifies the layer for the various width and spacing values. This layer must be a routing layer.

minSpacing

Optional argument that specifies the recommended minimum spacing for *layerName* for routes using this NONDEFAULTRULE to other geometries.

width

Specifies the required minimum width for layerName.

wireExtension

Optional argument that specifies the distance by which wires are extended at vias.The value must be greater than or equal to half of the routing width for the layer, as defined in the nondefault rule. Specify 0 to ignore this argument.

resistance

This argument is obsolete. Specify 0 to ignore this argument.

capacitance

This argument is obsolete. Specify 0 to ignore this argument.

edgecap

This argument is obsolete. Specify 0 to ignore this argument.

lefwNonDefaultRuleMinCuts

Wries a MINCUTS statement in the NONDEFAULTRULE statement. The MINCUTS statement is optional and can be used more than once in a NONDEFAULTRULE statement.

Syntax

```
lefwNonDefaultRuleMinCuts(
     const char* layerName,
     int numCuts)
```

Arguments

layerName

Specifies the cut layer.

numCuts

Specifies the minimum number of cuts allowed for any via using *layerName*.

lefwNonDefaultRuleStartVia

Starts a `VIA` statement in the `NONDEFAULTRULE` statement. The `VIA` statement is optional and can be used more than once in a `NONDEFAULTRULE` statement.

Each `VIA` statement must start and end with the `lefwNonDefaultRuleStartVia` and `lefwNonDefaultRuleEndVia` routines. The following routines can be included within a `VIA` statement:

- lefwViaLayer on page 241
- lefwViaLayerPolygon on page 242
- lefwViaLayerRect on page 243
- lefwViaResistance on page 243
- lefwViaViarule on page 243 (and its related routines)

Syntax

```
lefwNonDefaultRuleStartVia(
     const char* viaName,
     const char* isDefault)
```

Arguments

viaName

Specifies the name for the via.

isDefault

Identifies the via as the default via between the specified layers.

NULL	Ignores the argument.
DEFAULT	Identifies the via as the default via.

lefwNonDefaultRuleEndVia

Ends the VIA statement for the specified *viaName*. Each VIA statement must start and end with the lefwNonDefaultRuleStartVia and lefwNonDefaultRuleEndVia routines.

Syntax

```
lefwNonDefaultRuleEndVia(
     const char* viaName)
```

lefwNonDefaultRuleUseVia

Writes a USEVIA statement in a NONDEFAULTRULE statement. The USEVIA statement is optional and can be used more than once in a NONDEFAULTRULE statement.

Syntax

```
lefwNonDefaultRuleUseVia(
     const char* viaName)
```

Arguments

viaName

Specifies a previously defined via from the LEF VIA statement, or a previously defined NONDEFAULTRULE via to use with this routing rule.

lefwNonDefaultRuleUseViaRule

Wrties a USEVIARULE statement in the NONDEFAULTRULE statement. The USEVIARULE statement is optional and can be used more than once in a NONDEFAULTRULE statement.

Syntax

```
lefwNonDefaultRuleUseViaRule(
     const char* viaRuleName)
```

Arguments

viaRuleName

Specifies a previously defined VIARULE GENERATE rule to use with this routing rule. You cannot specify a rule from a VIARULE without a GENERATE keyword.

Nondefault Rules Example

The following example shows a callback routine with the type lefwNonDefaultCbkType. This example does not include information on how to create a via within the nondefault rule. For an example of how to create a via, see the Via section.

```
int nonDefaultCB (lefwCallbackType_e type,
                       lefiUserData userData) {
         int    res;

         // Check if the type is correct
         if (type != lefwNonDefaultCbkType) {
             printf("Type is not lefwNonDefaultCbkType, terminate
               writing.\n");
             return 1;
         }

         res = lefwStartNonDefaultRule("RULE1");
         CHECK_RES(res);
         res = lefwNonDefaultRuleLayer("RX", 10.0, 2.2, 6);
         CHECK_RES(res);
         res = lefwNonDefaultRuleLayer("PC", 10.0, 2.2, 0);
         CHECK_RES(res);
         res = lefwEndNonDefaultRule("RULE1");
         CHECK_RES(res);

         return 0;}
```

Property

The Property routines write a LEF PROPERTY statement in a VIA, VIARULE, LAYER, MACRO, or NONDEFAULTRULE section. The PROPERTY statement is optional and can be used more than once in these sections.

For examples of the routines described here, see "Property Example" on page 226.

All routines return `0` if successful.

lefwIntProperty

Writes a `PROPERTY` statement that defines a named property with an *`integer`* value. The `PROPERTY` statement is optional and can be used more than once in a LEF file.

Syntax

```
int lefwIntProperty(
     const char* propName,
     int propValue)
```

Arguments

propName

Specifies the name of the property.

propValue

Specifies an integer value.

lefwRealProperty

Writes a `PROPERTY` statement that defines a named property with a *`real`* number value. The `PROPERTY` statement is optional and can be used more than once in a LEF file.

Syntax

```
int lefwRealProperty(
     const char* propName,
     double propValue)
```

Arguments

propName

Specifies the name of the property.

propValue

Specifies a real value.

lefwStringProperty

Writes a PROPERTY statement that defines a named property with a *string* value. The PROPERTY statement is optional and can be used more than once in a LEF file.

Syntax

```
int lefwStringProperty(
     const char* propName,
     const char* propValue)
```

Arguments

propName

Specifies the name of the property.

propValue

Specifies a string value.

Property Example

The following example shows how to create property inside a Macro callback routine. It can be used for Layer, Via, Via Rule, Via within the Nondefault Rule, and Macro.

```
int macroCB (lefwCallbackType_e type,
                  lefiUserData userData) {

          int res;

          ...
          res = lefwStringProperty("TYPE", "special");
          CHECK_RES(res);
          res = lefwIntProperty("intProp", 23);
          CHECK_RES(res);
          res = lefwRealProperty("realProp", 24.25);
          CHECK_RES(res);
          ...

          return 0;}
```

Property Definitions

Property Definitions routines write a LEF `PROPERTYDEFINTIONS` statement. The `PROPERTYDEFINTIONS` statement is optional and can be used only once in a LEF file. For syntax information about the LEF `PROPERTYDEFINTIONS` statement, see "Property Definitions" in the *LEF/DEF Language Reference*.

You must use the `lefwStartPropDef` and `lefwEndPropDef` routines to start and end the `PROPERTYDEFINTIONS` statement. The `lefwPropDef` routine must be included between these routines.

For examples of the routines described here, see "Property Definitions Examples" on page 230.

All routines return `0` if successful.

lefwStartPropDef

Starts the `PROPERTYDEFINTIONS` statement.

Syntax

```
int lefwStartPropDef()
```

lefwEndPropDef

Ends the `PROPERTYDEFINTIONS` statement.

Syntax

```
int lefwEndPropDef(
```

lefwIntPropDef

Writes an integer property definition in the `PROPERTYDEFINITIONS` statement. The `lefwIntProperty` routine is optional and can be used more than once in a `PROPERTYDEFINITIONS` statement.

Syntax

```
int lefwIntPropDef(
     const char* objType,
     const char* propName,
     double leftRange,
     double rightRange,
     int propValue)
```

Arguments

leftRange rightRange

Optional arguments that limit integer property values to a specified range. Specify `0` to ignore these arguments.

objType

Specifies the object type for which you are defining properties.
Value: `LIBRARY`, `LAYER`, `VIA`, `VIARULE`, `NONDEFAULTRULE`, `MACRO`, or `PIN`

propName

Specifies a unique property name for the object type.

propValue

Optional argument that specifies an integer value for an object type. Specify `NULL` to ignore this argument.

lefwRealPropDef

Writes a real property definition in the `PROPERTYDEFINITIONS` statement. The `lefwRealPropDef` routine is optional and can be used more than once in a `PROPERTYDEFINITIONS` statement.

Syntax

```
int lefwRealPropDef(
     const char* objType,
     const char* propName,
     double leftRange,
     double rightRange,
     int propValue)
```

Arguments

leftRange rightRange

Optional arguments that limit real property values to a specified range. Specify `0` to ignore these arguments.

objType

Specifies the object type for which you are defining properties.
Value: `LIBRARY`, `LAYER`, `VIA`, `VIARULE`, `NONDEFAULTRULE`, `MACRO`, or `PIN`

propName

Specifies a unique property name for the object type.

propValue

Optional argument that specifies a real value for an object type. Specify `NULL` to ignore this argument.

lefwStringPropDef

Writes a string property definition in the `PROPERTYDEFINITIONS` statement. The `lefwStringPropDef` routine is optional and can be used more than once in a `PROPERTYDEFINITIONS` statement.

Syntax

```
int lefwStringPropDef(
     const char* objType,
     const char* propName,
     double leftRange,
     double rightRange,
     int propValue)
```

Arguments

leftRange rightRange

Optional arguments that limit property values to a specified range. Specify `0` to ignore these arguments.

objType

Specifies the object type for which you are defining properties.
Value: `LIBRARY`, `LAYER`, `VIA`, `VIARULE`, `NONDEFAULTRULE`, `MACRO`, or `PIN`

propName

Specifies a unique property name for the object type.

propValue

Optional argument that specifies a string value for an object type. Specify `NULL` to ignore this argument.

Property Definitions Examples

The following example shows a callback routine with the type `lefwPropDefCbkType`. This example does not show all of the combinations of Property Definitions defined.

```
int propDefCB (lefwCallbackType_e type,
                    lefiUserData userData) {
         int res;

         // Check if the type is correct
         if (type != lefwPropDefCbkType) {
             printf("Type is not lefwPropDefCbkType, terminate
               writing.\n");
             return 1;
         }
         res = lefwStartPropDef();
         CHECK_RES(res);
         res = lefwStringPropDef("LIBRARY", "NAME", 0, 0,
          "Cadence96");
         CHECK_RES(res);
         res = lefwIntPropDef("LIBRARY", "intNum", 0, 0, 20);
         CHECK_RES(res);
         res = lefwRealPropDef("LIBRARY", "realNum", 0, 0, 21.22);
         CHECK_RES(res);
         res = lefwEndPropDef();
         CHECK_RES(res);

         return 0;}
```

Same-Net Spacing

Same-Net Spacing routines write a LEF `SPACING` statement. The `SPACING` statement is optional and can be used only once in a LEF file. For syntax information about the LEF `SPACING` statement, see "Same-Net Spacing" in the *LEF/DEF Language Reference*.

You must use the `lefwStartSpacing` and `lefwEndSpacing` routines to start and end the `SPACING` statement. The `lefwSpacing` routine must be included between these routines.

For examples of the routines described here, see "Same-Net Spacing Examples" on page 232.

All routines return `0` if successful.

lefwStartSpacing

Writes the `SPACING` statement.

Syntax

```
int lefwStartSpacing()
```

lefwEndSpacing

Ends the `SPACING` statement.

Syntax

```
int lefwEndSpacing()
```

lefwSpacing

Writes the `SAMENET` statement. The `SAMENET` statement is required and can be used more than once.

Syntax

```
int lefwSpacing(
     const char* layerName1,
     const char* layerName2,
     double minSpace,
     const char* stack)
```

Arguments

`layerName1`, *`layerName2`*

Specify the names of the layers for which the same-net spacing rule applies. You can specify spacing rules for routing layers and cut layers. For a routing layer, the same-net spacing rule is defined by specifying the same layer name twice.

minSpace

Specifies the minimum spacing.

stack

Optional argument that allows stacked vias at a routing layer. Specify NULL to ignore this argument.

Same-Net Spacing Examples

The following example shows a callback routine with the type lefwSpacingCbkType.

```
int spacingCB (lefwCallbackType_e type,
                    lefiUserData userData) {
         int res;

         // Check if the type is correct
         if (type != lefwSpacingCbkType) {
             printf("Type is not lefwSpacingCbkType, terminate
               writing.\n");
             return 1;
         }

         res = lefwStartSpacing();
         CHECK_RES(res);
         res = lefwSpacing("CUT01", "CA", 1.5, NULL);
         CHECK_RES(res);
         res = lefwEndSpacing();
         CHECK_RES(res);

         return 0;}
```

Site

The Site routines write a LEF SITE statement. The SITE statement is optional and can be used more than once in a LEF file. For syntax information about the LEF SITE statement, see "Site" in the *LEF/DEF Language Reference*.

Each SITE statement must be defined with a lefwSite and lefwEndSite routine.

All routines return 0 if successful.

lefwSite

Writes a SITE statement.

Syntax

```
int lefwSite(
     const char* siteName,
     const char* classType,
     const char* symmetry,
     double width,
     double height)
```

Arguments

classType

Specifies whether the site is a core site or an I/O pad site.
Value: PAD or CORE.

siteName

Specifies the name of the placement site.

symmetry

Specifies how the site is symmetrical in normal orientation.
Value: Specify one of the following:

X	Defines the site as symmetric about the x axis.
Y	Defines the site as symmetric about the y axis.
R90	Defines the site as symmetric when rotated 90 degrees.

width, *height*

Specify the dimensions of the site in normal (or north) orientation, in microns.

lefwEndSite

Ends a SITE statement.

Syntax

```
int lefwEndSite(
     const char* siteName)
```

Arguments

siteName

Specifies the name of the placement site.

lefwSiteRowPattern

Writes a `ROWPATTERN` statement in the SITE statement. The `ROWPATTERN` statement is optional and can be used more than once in a `SITE` statement.

Syntax

```
lefwSiteRowPattern( const char* siteName,
     int orient)
```

Arguments

siteName

Specifies the name of a previously defined site.

orient

Specifies the orientation for the previously defined site.
Value: `0` to `7`. For more information, see "Orientation Codes" *on page 21*.

lefwSiteRowPatternStr

Also writes a `ROWPATTERN` statement. This routine is the same as the `lefwSiteRowPattern` routine, with the exception of the *orient* argument, which takes a string instead of an integer. The `ROWPATTERN` statement is optional and can be used more than once in a `SITE` statement.

Syntax

```
lefwSiteRowPattern( const char* siteName,
     int orient)
```

Arguments

siteName

Specifies the name of a previously defined site.

orient

Specifies the orientation for the previously defined site.
Value: N, W, S, E, FN, FW, FS, or FE.

Site Examples

The following example shows a callback routine with the type lefwSiteCbkType.

```
int siteCB (lefwCallbackType_e type,
                 lefiUserData userData) {
         int    res;

         // Check if the type is correct
         if (type != lefwSiteCbkType) {
             printf("Type is not lefwSiteCbkType, terminate
               writing.\n");
             return 1;
         }

         res = lefwSite("CORE1", "CORE", "X", 67.2, 6);
         CHECK_RES(res);

         return 0;}
```

Units

Units routines write a LEF UNITS statement. The UNITS statement is optional and can be used only once in a LEF file. For syntax information about the LEF UNITS statement, see "Units" in the *LEF/DEF Language Reference*.

You must use the lefwStartUnits and lefwEndSpacing routines to start and end the UNITS statement. The lefwUnits routine must be included between these routines.

For examples of the routines described here, see "Units Examples" on page 237.

All routines return 0 if successful.

lefwStartUnits

Starts the UNITS statement.

Syntax

```
int lefwStartUnits()
```

lefwEndUnits

Ends the `UNITS` statement.

Syntax

```
int lefwEndUnits()
```

lefwUnits

Writes a `UNITS` statement. The `UNITS` statement is required whenever the `lefwStartSpacing` routine is specified.

Syntax

```
int lefwUnits(
     double time,
     double capacitance,
     double resistance,
     double power,
     double current,
     double voltage,
     double database)
```

Arguments

time

Optional argument that specifies a `TIME NANOSECONDS` statement. This interprets one LEF time unit as one nanosecond. Specify `0` to ignore this argument.

capacitance

Optional argument that specifies a `CAPACITANCE PICOFARADS` statement. This interprets one LEF capacitance unit as one picofarad. Specify `0` to ignore this argument.

resistance

Optional argument that specifies a `RESISTANCE OHMS` statement. This interprets one LEF resistance unit as one ohm. Specify `0` to ignore this argument.

power

Optional argument that specifies a `POWER MILLIWATTS` statement. This interprets one LEF power unit as one milliwatt. Specify `0` to ignore this argument.

current

Optional argument that specifies a `CURRENT MILLIAMPS` statement. This interprets one LEF current unit as one milliamp. Specify `0` to ignore this argument.

voltage

Optional argument that specifies a `VOLTAGE VOLTS` statement. This interprets one LEF voltage unit as one volt. Specify `0` to ignore this argument.

database

Optional argument that specifies a `DATABASE MICRONS` statement. This interprets one LEF distance unit as multiplied when converted into database units. Specify `0` to ignore this argument.

lefwUnitsFrequency

Writes a `FREQUENCY` statement in the `UNITS` statement. The `FREQUENCY` statement is optional and can be used only once in a `UNITS` statement.

Syntax

```
int lefwUnitsFrequency(
     double frequency)
```

Arguments

frequency

Specifies a `FREQUENCY MEGAHERTZ` statement. This interprets one LEF frequency unit as one megahertz.

Units Examples

The following example shows a callback routine with the type `lefwUnitsCbkType`.

```
int unitsCB (lefwCallbackType_e type,
                  lefiUserData userData) {
         int res;

         // Check if the type is correct
```

```
    if (type != lefwUnitsCbkType) {
        printf("Type is not lefwUnitsCbkType, terminate
          writing.\n");
        return 1;
    }
    res = lefwStartUnits();
    CHECK_RES(res);
    res = lefwUnits(100, 10, 10000, 10000, 10000, 1000, 0);
    CHECK_RES(res);
    res = lefwEndUnits();
    CHECK_RES(res);

    return 0;}
```

Use Min Spacing

The Use Min Spacing routine writes a LEF `USEMINSPACING` statement, which defines how minimum spacing is calculated for obstruction geometries. The `USEMINSPACING` statement is optional and can be used more than once in a LEF file.

For syntax information about the LEF `USEMINSPACING` statement, see "Use Min Spacing" in the *LEF/DEF Language Reference*.

This routine returns `0` if successful.

lefwUseMinSpacing

Writes a `USEMINSPACING` statement.

Syntax

```
int lefwUseMinSpacing(
     const char* type,
     const char* onOff)
```

Arguments

type

Specifies that the minimum spacing applies to obstruction geometries.
Value: `OBS`

onOff

Specifies how to calculate the minimum spacing.
Value: Specify one of the following:

`ON`	Spacing is computed as if the `MACRO OBS` shapes were min-width wires. Some LEF models abstract many min-width wires as a single large `OBS` shape; therefore using wide wire spacing would be too conservative.
`OFF`	Spacing is computed to `MACRO OBS` shapes as if they were actual routing shapes. A wide `OBS` shape would use wide wire spacing rules, and a thin `OBS` shapes would use thin wire spacing rules.

Version

The version routine writes a LEF `VERSION` statement. For syntax information about the LEF `VERSION` statement, see "Version" in the *LEF/DEF Language Reference*.

The `VERSION` statement is part of the LEF file header (which also includes the `BUSBITCHARS`, and `DIVIDERCHAR` statements). If the statements in the header section are not defined, many applications assume default values for them. However, the default values are not formally part of the language definition; therefore you cannot be sure that the same assumptions are used in all applications. You should always explicitly define these values.

This routine returns `0` if successful.

lefwVersion

Writes a `VERSION` statement. The `VERSION` statement can be used only once in a LEF file.

Syntax

```
int lefwVersion(
     int vers1,
     int vers2)
```

Arguments

vers1, *vers2*

Specify which version of the LEF syntax is being used. *vers1* is the major value. *vers2* is the minor value.

Version Examples

The following example shows a callback routine with the type lefwVersionCbkType.

```
int versionCB (lefwCallbackType_e type,
                    lefiUserData userData) {
         int res;

         // Check if the type is correct
         if (type != lefwVersionCbkType) {
             printf("Type is not lefwVersionCbkType, terminate
               writing.\n");
             return 1;
         }

         res = lefwVersion(5, 3);
         CHECK_RES(res);

         return 0;}
```

Via

Via routines write a LEF VIA section. A VIA section is optional and can be used more than once in a LEF file. For syntax information about the LEF VIA section, see "Via" in the *LEF/DEF Language Reference*.

Each VIA section must start and end with the lefwStartVia and lefwEndVia routines. The remaining via routines must be included between these routines.

In addition to the routines described in this section, you can include a PROPERTY statement within a VIA section. For more information about these routines, see “Property” on page 224.

For examples of the routines described here, see “Via Examples” on page 247.

All routines return 0 if successful.

lefwStartVia

Starts a `VIA` section.

Syntax

```
int lefwStartVia(
     const char* viaName,
     const char* isDefault)
```

Arguments

viaName

Specifies the name for the via.

isDefault

Optional argument that identifies the via as the default via between the specified layers.

`NULL`	Ignores the argument.
`DEFAULT`	Identifies the via as the default via.

lefwEndVia

Ends the `VIA` section for the specified *viaName* value.

Syntax

```
int lefwEndVia(
     const char* viaName)
```

lefwViaLayer

Writes a `LAYER` statement for a via. Either a `LAYER` or a `VIARULE` statement is required in a `VIA` section. A `LAYER` statement can be used more than once for a via.

If you specify this routine, you must also specify one of the following routines:

- lefwViaLayerPolygon on page 242
- lefwViaLayerRect on page 243

You can also optionally specify the following routine:

- lefwViaResistance on page 243

Syntax

```
int lefwViaLayer(
     const char* layerName)
```

Arguments

layerName

Specifies the layer on which to create the rectangles that make up the via. Normal vias have exactly three layers used: a cut layer and two layers that touch the cut layer (routing or masterslice).

lefwViaLayerPolygon

Writes a `POLYGON` statement for a via. Either a `POLYGON` or `RECT` statement is required if a `LAYER` statement is specified in a `VIA` section, and can be used more than once.

Syntax

```
lefwViaLayerPolygon(
     int num_polys,
     double* xl,
     double* yl)
```

Arguments

num_polys

Specifies the number of polygon sides.

xl yl

Specifies a sequence of points to generate a polygon geometry. The polygon edges must be parallel to the x axis, the y axis, or at a 45-degree angle. The polygon is generated by connecting each successive point, then connecting the first and last points.

lefwViaLayerRect

Writes a `RECT` statement. Either a `POLYGON` or `RECT` statement is required if a `LAYER` statement is specified in a `VIA` section, and can be used more than once.

Syntax

```
int lefwViaLayerRect(
     double x1l,
     double y1l,
     double x2l,
     double y2l)
```

Arguments

x1l, *y1l*, *x2l*, *y2l*

Specify the points that make up the via.

lefwViaResistance

Writes a `RESISTANCE` statement. The `RESISTANCE` statement is optional and can be used only once with a `LAYER` statement in a `VIA` section.

Syntax

```
int lefwViaResistance(
     double resistance)
```

Arguments

resistance

Specifies the total resistance of the via, in units of ohms, given as the resistance per via. Note that this is not a resistance per via-cut value; it is the total resistance of the via.

lefwViaViarule

Writes a `VIARULE` statement for the via. Either a `LAYER` or a `VIARULE` statement is required in a `VIA` section. A `VIARULE` statement can be used only once in a `VIA` section.

If you specify this routine, you can optionally specify the following routines:

- lefwViaViaruleOffset on page 245
- lefwViaViaruleOrigin on page 245
- lefwViaViarulePattern on page 246
- lefwViaViaruleRowCol on page 246

Syntax

```
lefwViaViarule(
     const char* viaRuleName,
     double xCutSize,
     double yCutSize,
     const char* botMetalLayer,
     const char* cutLayer,
     const char* topMetalLayer,
     double xCutSpacing,
     double yCutSpacing,
     double xBotEnc,
     double yBotEnc,
     double xTopEnc,
     double yTopEnc)
```

Arguments

viaRuleName

Specifies the name of the LEF VIARULE that produced this via. This name must refer to a previouslydefined VIARULE GENERATE rule name. This indicates that the via is the result of automatic via generation, and that the via name is only used locally inside this LEF file.

xCutSize yCutSize

Specifies the required width (*xSize*) and height (*ySize*) of the cut layer rectangles.

botMetalLayer cutLayer topMetalLayer

Specifies the required names of the bottom routing layer, cut layer, and top routing layer. These layers must be previously defined in layer definitions, and must match the layer names defined in the specified LEF *viaRuleName*.

xCutSpacing yCutSpacing

Specifies the required x and y spacing between cuts. The spacing is measured between one cut edge and the next cut edge.

xBotEnc yBotEnc xTopEnc yTopEnc

Specifies the required x and y enclosure values for the bottom and top metal layers. The enclosure measures the distance from the cut array edge to the metal edge that encloses the cut array.

lefwViaViaruleOffset

Writes an `OFFSET` statement for the via. The `OFFSET` statement is optional with a `VIARULE` statement and can be used only once in a `VIA` section.

Syntax

```
lefwViaViaruleOffset(
     double xBotOffset,
     double yBotOffset,
     double xTopOffset,
     double yTopOffset)
```

Arguments

xBotOffset yBotOffset xTopOffset yTopOffset

Specifies the x and y offset for the bottom and top metal layers. By default, the `0,0` origin of the via is the center of the cut array and the enclosing metal rectangles. These values allow each metal layer to be offset independently.

After the non-shifted via is computed, the metal layer rectangles are offset by adding the appropriate values—the x/y *BotOffset* values to the metal layer below the cut layer, and the x/ y *TopOffset* values to the metal layer above the cut layer. These offsets are in addition to any offset caused by the *ORIGIN* values.

lefwViaViaruleOrigin

Writes an `ORIGIN` statement for the via. The `ORIGIN` statement is optional with a `VIARULE` statement and can be used only once in a `VIA` section.

Syntax

```
lefwViaViaruleOrigin(
     double xOffset,
     double yOffset)
```

Arguments

xOffset yOffset

Specifies the x and y offset for all of the via shapes. By default, the `0,0` origin of the via is the center of the cut array and the enclosing metal rectangles. After the non-shifted via is computed, all cut and metal rectangles are offset by adding these values.

lefwViaViarulePattern

Writes a `PATTERN` statement for the via. The `PATTERN` statement is optional with a `VIARULE` statement and can be used only once in a `VIA` section.

Syntax

```
lefwViaViarulePattern(
     const char* cutPattern)
```

Arguments

cutPattern

Specifies the cut pattern encoded as an ASCII string. This parameter is only required when some of the cuts are missing from the array of cuts, and defaults to "all cuts are present," if not specified.

lefwViaViaruleRowCol

Writes a `ROWCOL` statement for the via. The `ROWCOL` statement is optional with a `VIARULE` statement and can be used only once in a `VIA` section.

Syntax

```
lefwViaViaRuleRowCol(
     int numCutRows,
     int numCutCols)
```

Arguments

numCutRows numCutCols

Specifies the number of cut rows and columns that make up the via array.

Via Examples

The following example shows a callback routine with the type `lefwViaCbkType`.

```
int viaCB (lefwCallbackType_e type,
               lefiUserData userData) {
         int res;

         // Check if the type is correct
         if (type != lefwViaCbkType) {
             printf("Type is not lefwViaCbkType, terminate
               writing.\n");
             return 1;
         }

         res = lefwStartVia("RX_PC", "DEFAULT");
         CHECK_RES(res);
         res = lefwViaResistance(2);
         CHECK_RES(res);
         res = lefwViaForeign("IN1X", 0, 0, -1);
         CHECK_RES(res);
         res = lefwViaLayer("RX");
         CHECK_RES(res);
         res = lefwViaLayerRect(-0.7, -0.7, 0.7, 0.7);
         CHECK_RES(res);
         res = lefwViaLayer("CUT12");
         CHECK_RES(res);
         res = lefwViaLayerRect(-0.25, -0.25, 0.25, 0.25);
         CHECK_RES(res);
         res = lefwRealProperty("realProperty", 32.33);
         CHECK_RES(res);
         res = lefwIntProperty("COUNT", 34);
         CHECK_RES(res);
         res = lefwEndVia("RX_PC");
         CHECK_RES(res);

         return 0;}
```

Via Rule

Via Rule routines write a LEF VIARULE statement. A VIARULE or a VIARULE GENERATE statement is required in a LEF file. You can create more than one VIARULE statement in a LEF file. For syntax information about the LEF VIARULE statement, see "Via Rule" in the *LEF/DEF Language Reference*.

You must use the `lefwStartViaRule` and `lefwEndViaRule` routines to start and end the `VIARULE` statement. The `lefwViaRuleLayer` and `lefwViaRuleVia` routines must be included between these routines.

For examples of the routines described here, see "Via Rule Examples" on page 250.

In addition to the routines described in this section, you can include a `PROPERTY` statement within a `VIARULE` statement. For more information about these routines, see "Property" on page 224.

All routines return `0` if successful.

lefwStartViaRule

Starts a `VIARULE` statement.

Syntax

```
int lefwStartViaRule(
     const char* viaRuleName)
```

Arguments

viaRuleName

Specifies the name to identify the via rule.

lefwEndViaRule

Ends the `VIARULE` statement for the specified *viaRuleName* value.

Syntax

```
int lefwEndViaRule(
     const char* viaRuleName)
```

lefwViaRuleLayer

Writes a `LAYER` statement. The `LAYER` statement is required and must be used exactly twice in a `VIARULE` statement.

Syntax

```
int lefwViaRuleLayer(
     const char* layerName,
     const char* direction,
     double minWidth,
     double maxWidth,
     double overhang,
     double metalOverhang)
```

Arguments

layerName

Specifies the top or bottom routing layer of the via.

direction

Specifies the wire direction. If you specify a width range, the rule applies to wires of the specified *direction* that fall within the range. Otherwise, the rule applies to all wires on the layer of the specified *direction*.
Value: HORIZONTAL or VERTICAL

minWidth maxWidth

Optional arguments that specify a wire width range within which the wire must fall in order for the rule to apply. That is, the wire width must be greater than or equal to *minWidth* and less than or equal to *maxWidth*. Specify 0 to ignore these arguments.

overhang

This argument is obsolete. Specify 0 to ignore this argument.

metalOverhang

This argument is obsolete. Specify 0 to ignore this argument.

lefwViaRuleVia

Writes a VIA statement. The VIA statement is required and can be used more than once after both lefwViaRuleLayer routines are used.

Syntax

```
int lefwViaRuleVia(
     const char* viaName)
```

Arguments

viaName

Specifies a previously defined via to test for the current via rule. The first via in the list that can be placed at the location without design rule violations is selected. The vias must all have exactly three layers in them. The three layers must include the same routing layers as listed in the `LAYER` statements of the `VIARULE`, and a cut layer that is between the two routing layers.

Via Rule Examples

The following example shows a callback routine with the type `lefwViaRuleCbkType`.

```
int viaRuleCB(lefwCallbackType_e c, lefiUserData ud) {
         int    res;

         // Check if the type is correct
         if (type != lefwViaCbkType) {
             printf("Type is not lefwViaCbkType, terminate
               writing.\n");
             return 1;
         }

         res = lefwStartViaRule("VIALIST12");
         CHECK_RES(res);
         lefwAddComment("Break up the old lefwViaRule into 2
          routines");
         lefwAddComment("lefwViaRuleLayer and lefwViaRuleVia");
         res = lefwViaRuleLayer("M1", "VERTICAL", 9.0, 9.6, 4.5,
          0);
         CHECK_RES(res);
         res = lefwViaRuleLayer("M2", "HORIZONTAL", 3.0, 3.0, 0,
          0);
         CHECK_RES(res);
         res = lefwViaRuleVia("VIACENTER12");
         CHECK_RES(res);
         res = lefwStringProperty("vrsp", "new");
         CHECK_RES(res);
         res = lefwIntProperty("vrip", 1);
         CHECK_RES(res);
         res = lefwRealProperty("vrrp", 4.5);
         CHECK_RES(res);
         res = lefwEndViaRule("VIALIST12");
         CHECK_RES(res);

         return 0;}
```

Via Rule Generate

The Via Rule Generate routines write a LEF `VIARULE GENERATE` statement. A `VIARULE GENERATE` or a `VIARULE` statement is required in a LEF file. You can create more than one `VIARULE GENERATE` statement in a LEF file. For syntax information the LEF `VIARULE GENERATE` statement, see "Via Rule Generate" in the *LEF/DEF Language Reference*.

You must use the `lefwStartViaRuleGen` and `lefwEndViaRuleGen` routines to start and end the `VIARULE GENERATE` statement. All other routines must be included between these routines.

Use the Via Rule Generate routines to cover special wiring that is not explicitly defined in the Via Rule routines.

All routines return `0` if successful.

lefwStartViaRuleGen

Starts a `VIARULE GENERATE` statement.

Syntax

```
int lefwStartViaRuleGen(
     const char* viaRuleName)
```

Arguments

viaRuleName
Specifies the name for the via rule (formula).

lefwEndViaRuleGen

Ends the `VIARULE GENERATE` statement for the specified *viaRuleName* value.

Syntax

```
int lefwEndViaRuleGen(
     const char* viaRuleName)
```

lefwViaRuleGenDefault

Writes a DEFAULT statement for the via.The DEFAULT statement specifies that the via rule can be used to generate vias for the default routing rule, and to supplement any DEFAULT fixed vias that might be predefined in the LEF VIA statement, as the router needs them. The DEFAULT statement is optional and can be used only once for a VIARULE GENERATE statement.

Syntax

```
lefwViaRuleGenDefault()
```

lefwViaRuleGenLayer

Writes a routing LAYER statement. Either the routing LAYER statement or the ENCLOSURE statement is required and must be used exactly twice in a VIARULE GENERATE statement.

Syntax

```
int lefwViaRuleGenLayer(
     const char* layerName,
     const char* direction,
     double minWidth,
     double maxWidth,
     double overhang,
     double metalOverhang)
```

Arguments

layerName

Specifies the routing layer for the top or bottom of the via.

direction

Specifies the wire direction. If you specify a width range, the rule applies to wires of the specified *direction* that fall within the range. Otherwise, the rule applies to all wires on the layer of the specified *direction*.
Value: HORIZONTAL or VERTICAL

minWidth maxWidth

Optional arguments that specify a wire width range within which the wire must fall in order for the rule to apply. That is, the wire width must be greater than or equal to *minWidth* and less than or equal to *maxWidth*. Specify 0 to ignore these arguments.

overhang

This argument is obsolete. Specify `0` to ignore this argument.

metalOverhang

This argument is obsolete. Specify `0` to ignore this argument.

lefwViaRuleGenLayer3

Writes a cut `LAYER` statement. The cut `LAYER` statement is required and can be used only once after either both `lefwViaRuleGenLayer`, or both `lefwViaRuleGenLayerEnclosure` routines are used.

Syntax

```
int lefwViaRuleGenLayer3(
     const char* layerName,
     double xl,
     double yl,
     double xh,
     double yh,
     double xSpacing,
     double ySpacing,
     double resistance)
```

Arguments

layerName

Specifies the cut layer for the generated via.

xl yl xh yh

Specifies the location of the lower left contact cut rectangle.

xSpacing ySpacing

Defines center-to-center spacing in the x and y dimensions to create an array of contact cuts.The number of cuts of an array in each direction is the most that can fit within the bounds of the intersection formed by the two special wires. Cuts are only generated where they do not violate stacked or adjacent via design rules.

resistance

Optional argument that specifies the resistance of the cut layer, given as the resistance per contact cut. Specify `0` to ignore this argument.

lefwViaRuleGenLayerEnclosure

Writes an `ENCLOSURE` statement. Either the `ENCLOSURE` statement or the routing `LAYER` statement is required and must be used exactly twice in a `VIARULE GENERATE` statement.

Syntax

```
int lefwViaRuleGenLayerEnclosure(
     const char* layerName,
     double overhang1,
     double overhang2,
     double minWidth,
     double maxWidth)
```

Arguments

layerName

Specifies the routing layer for the top or bottom of the via.

overhang1 overhang2

Specifies that the via must be covered by metal on two opposite sides by at least *overhang1*, and on the other two sides by at least *overhang2*. The via generation code then chooses the direction of overhang that best maximizes the number of cuts that can fit in the via.

minWidth maxWidth

Optional arguments that specify a wire width range within which the wire must fall in order for the rule to apply. That is, the wire width must be greater than or equal to *minWidth* and less than or equal to *maxWidth*. Specify `0` to ignore this argument.

Via Rule Generate Examples

The following example shows a callback routine with the type `lefwViaRuleCbkType` with Generate.

```
int viaRuleCB(lefwCallbackType_e c, lefiUserData ud) {
          int    res;

          // Check if the type is correct
          if (type != lefwViaCbkType) {
              printf("Type is not lefwViaCbkType, terminate
                writing.\n");
              return 1;
          }
```

```
res = lefwStartViaRuleGen("VIAGEN12");
CHECK_RES(res);
res = lefwViaRuleGenLayer("M1", "VERTICAL", 0.1, 19, 1.4,
 0);
CHECK_RES(res);
res = lefwViaRuleGenLayer("M2", "HORIZONTAL", 0, 0, 1.4,
 0);
CHECK_RES(res);
res = lefwViaRuleGenLayer3("V1", -0.8, -0.8, 0.8, 0.8,
 5.6, 6.0, 0.2);
CHECK_RES(res);
res = lefwEndViaRuleGen("VIAGEN12");
CHECK_RES(res);

return 0;}
```

7

LEF Compressed File Routines

The Cadence® Library Exchange Format (LEF) reader provides the following routines for opening and closing compressed LEF files. These routines are used instead of the `fopen` and `fclose` routines that are used for regular LEF files.

lefGZipOpen

Opens a compressed LEF file. If the file opens with no errors, this routine returns a pointer to the file.

Syntax

```
lefGZFile lefGZipOpen(
     const char* gzipFile,
     const char* mode);
```

Arguments

gzipFile

Specifies the compressed file to open.

mode

Specifies how to open the file. Compressed files should be opened as read only; therefore, specify `"r"`.

lefGZipClose

Closes the compressed LEF file. If the file closes with no errors, this routine returns zero.

Syntax

```
int lefGZipClose(
     lefGZFile filePtr) ;
```

Arguments

filePtr

Specifies a pointer to the compressed file to close.

Example

The following example uses the `lefGZipOpen` and `lefGZipClose` routines to open and close a compressed file.

```
lefrInit() ;

for (fileCt = 0; fileCt < numInFile; fileCt++) {
    lefrReset();

    // Open the compressed LEF file for the reader to read
    if ((f = lefGZipOpen(inFile[fileCt], "r")) == 0) {
       fprintf(stderr, "Couldn't open input file '%s'\n", inFile[fileCt]);
       return(2) ;
    }

    (void)lefrEnableReadEncrypted();

    // Initialize the lef writer. Needs to be called first.
    status = lefwInit(fout);
    if (status != LEFW_OK)
       return 1;

    res = lefrRead((FILE*)f, inFile[fileCt], (void*)userData);

    if (res)
       fprintf(stderr, "Reader returns bad status.\n", inFile[fileCt]);

    // Close the compressed LEF file.
    lefGZipClose(f);
    (void)lefrPrintUnusedCallbacks(fout);
```

```
}
fclose(fout);

return 0;}
```

8

LEF File Comparison Utility

The Cadence® Library Exchange Format (LEF) reader provides the following utility for comparing LEF files.

lefdefdiff

Compares two LEF or DEF files and reports any differences between them.

Because LEF and DEF files can be very large, the `lefdefdiff` utility writes each construct from a file to an output file in the `/tmp` directory. The utility writes the constructs using the format:

section_head/*subsection*/*subsection*/ ... /*statement*

The `lefdefdiff` utility then sorts the output files and uses the `diff` program to compare the two files. Always verify the accuracy of the `diff` results.

Note: You must specify the `-lef` or `-def`, *inFileName1*, and *inFileName2* arguments in the listed order. All other arguments can be specified in any order after these arguments.

Syntax

```
lefdefdiff
     {-lef | -def}
     inFileName1
     inFileName2
     [-o outFileName]
     [-path pathName]
     [-quick]
     [-d]
     [-ignorePinExtra]
     [-ignoreRowName]
     [-h]
```

Arguments

`-d`

Uses the gnu `diff` program to compare the files for a smaller set of differences. Use this argument only for UNIX platforms.

`-h`

Returns the syntax and command usage for the `lefdefdiff` utility.

`-ignorePinExtra`

Ignores any `.extra`*N* statements in the pin name. This argument can only be used when comparing DEF files.

`-ignoreRowName`

Ignores the row name when comparing `ROW` statements in the DEF files. This argument can only be used when comparing DEF files.

inFileName1

Specifies the first LEF or DEF file.

inFileName2

Specifies the LEF or DEF file to compare with the first file.

`-lef | -def`

Specifies whether you are comparing LEF or DEF files.

`-o` *outFileName*

Outputs the results of the comparison to the specified file.
Default: Outputs the results to the screen

`-path` *pathName*

Temporarily stores the intermediate files created by the `lefdefdiff` utility in the specified path directory.
Default: Temporarily stores the files in the current directory

`-quick`

Uses the `bdiff` program to perform a faster comparison.

Example

The following example shows an output file created by the `lefdefdiff` utility after comparing two LEF files:

```
#The names of the two LEF files that were compared.
< file1.lef
> file2.lef
#Statements listed under Added were found in file2.lef but not in file1.lef.
Added:
> LAYER M1 SPACING 0.6
#Statements listed under Deleted were found in file1.lef but not in file2.lef.
Deleted:
< LAYER RX LENGTHTHRESHOLD 0.45
< LAYER RX LENGTHTHRESHOLD 0.9
< LAYER RX MINIMUMCUT 2 WIDTH 2.5
#Changed always contains two statements: the statement as it appears in
file1.lef and the statement as it appears in file2.lef.
CHANGED:
< MACRO INV_B EEQ INV SYMMETRY X Y R90
---
> MACRO INV_B CLASS CORE EEQ INV SYMMETRY X Y R90
Added:
> MACRO INV_B ORIGIN ( 0 0 )
Added:
> OBS PATH ( 58.8 3 ) ( 58.8 123 )
```

A

LEF Reader and Writer Examples

This appendix contains examples of the Cadence® Library Exchange Format (LEF) reader and writer.

- LEF Reader Program
- LEF Writer Program on page 320

LEF Reader Program

```
#ifdef WIN32
#pragma warning (disable : 4786)
#endif

#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <malloc.h>

#ifndef WIN32
#   include <unistd.h>
#else
#   include <windows.h>
#endif /* not WIN32 */
#include "lefrReader.hpp"
#include "lefwWriter.hpp"
#include "lefiDebug.hpp"
#include "lefiEncryptInt.hpp"
#include "lefiUtil.hpp"

char defaultName[128];
char defaultOut[128];
FILE* fout;
```

```
int printing = 0;     // Printing the output.
int parse65nm = 0;

// TX_DIR:TRANSLATION ON

void dataError() {
  fprintf(fout, "ERROR: returned user data is not correct!\n");
}

void checkType(lefrCallbackType_e c) {
  if (c >= 0 && c <= lefrLibraryEndCbkType) {
    // OK
  } else {
    fprintf(fout, "ERROR: callback type is out of bounds!\n");
  }
}

char* orientStr(int orient) {
  switch (orient) {
      case 0: return ((char*)"N");
      case 1: return ((char*)"W");
      case 2: return ((char*)"S");
      case 3: return ((char*)"E");
      case 4: return ((char*)"FN");
      case 5: return ((char*)"FW");
      case 6: return ((char*)"FS");
      case 7: return ((char*)"FE");
  };
  return ((char*)"BOGUS");
}

void lefVia(lefiVia* via) {
  int i, j;

  lefrSetCaseSensitivity(1);
  fprintf(fout, "VIA %s ", via->lefiVia::name());
  if (via->lefiVia::hasDefault())
     fprintf(fout, "DEFAULT");
  else if (via->lefiVia::hasGenerated())
     fprintf(fout, "GENERATED");
  fprintf(fout, "\n");
```

```
if (via->lefiVia::hasTopOfStack())
   fprintf(fout, "  TOPOFSTACKONLY\n");
if (via->lefiVia::hasForeign()) {
   fprintf(fout, "  FOREIGN %s ", via->lefiVia::foreign());
   if (via->lefiVia::hasForeignPnt()) {
      fprintf(fout, "( %g %g ) ", via->lefiVia::foreignX(),
              via->lefiVia::foreignY());
      if (via->lefiVia::hasForeignOrient())
         fprintf(fout, "%s ", orientStr(via->lefiVia::foreignOrient()));
   }
   fprintf(fout, ";\n");
}
if (via->lefiVia::hasProperties()) {
   fprintf(fout, "  PROPERTY ");
   for (i = 0; i < via->lefiVia::numProperties(); i++) {
      fprintf(fout, "%s ", via->lefiVia::propName(i));
      if (via->lefiVia::propIsNumber(i))
         fprintf(fout, "%g ", via->lefiVia::propNumber(i));
      if (via->lefiVia::propIsString(i))
         fprintf(fout, "%s ", via->lefiVia::propValue(i));
      switch (via->lefiVia::propType(i)) {
         case 'R': fprintf(fout, "REAL ");
                   break;
         case 'I': fprintf(fout, "INTEGER ");
                   break;
         case 'S': fprintf(fout, "STRING ");
                   break;
         case 'Q': fprintf(fout, "QUOTESTRING ");
                   break;
         case 'N': fprintf(fout, "NUMBER ");
                   break;
      }
   }
   fprintf(fout, ";\n");
}
if (via->lefiVia::hasResistance())
   fprintf(fout, "  RESISTANCE %g ;\n", via->lefiVia::resistance());
if (via->lefiVia::numLayers() > 0) {
   for (i = 0; i < via->lefiVia::numLayers(); i++) {
      fprintf(fout, "  LAYER %s\n", via->lefiVia::layerName(i));
      for (j = 0; j < via->lefiVia::numRects(i); j++)
```

```
          fprintf(fout, "    RECT ( %f %f ) ( %f %f ) ;\n",
                 via->lefiVia::xl(i, j), via->lefiVia::yl(i, j),
                 via->lefiVia::xh(i, j), via->lefiVia::yh(i, j));
       for (j = 0; j < via->lefiVia::numPolygons(i); j++) {
          struct lefiGeomPolygon poly;
          poly = via->lefiVia::getPolygon(i, j);
          fprintf(fout, "    POLYGON ");
          for (int k = 0; k < poly.numPoints; k++)
             fprintf(fout, " %g %g ", poly.x[k], poly.y[k]);
          fprintf(fout, ";\n");
       }
    }
  }
  if (via->lefiVia::hasViaRule()) {
     fprintf(fout, "  VIARULE %s ;\n", via->lefiVia::viaRuleName());
     fprintf(fout, "    CUTSIZE %g %g ;\n", via->lefiVia::xCutSize(),
                                        via->lefiVia::yCutSize());
     fprintf(fout, "    LAYERS %s %s %s ;\n", via->lefiVia::botMetalLayer(),
                    via->lefiVia::cutLayer(), via->lefiVia::topMetalLayer());
     fprintf(fout, "    CUTSPACING %g %g ;\n", via->lefiVia::xCutSpacing(),
                                        via->lefiVia::yCutSpacing());
     fprintf(fout, "    ENCLOSURE %g %g %g %g ;\n", via->lefiVia::xBotEnc(),
                     via->lefiVia::yBotEnc(), via->lefiVia::xTopEnc(),
                     via->lefiVia::yTopEnc());
     if (via->lefiVia::hasRowCol())
        fprintf(fout, "    ROWCOL %d %d ;\n", via->lefiVia::numCutRows(),
                                          via->lefiVia::numCutCols());
     if (via->lefiVia::hasOrigin())
        fprintf(fout, "    ORIGIN %g %g ;\n", via->lefiVia::xOffset(),
                                          via->lefiVia::yOffset());
     if (via->lefiVia::hasOffset())
        fprintf(fout, "    OFFSET %g %g %g %g ;\n", via->lefiVia::xBotOffset(),
                     via->lefiVia::yBotOffset(), via->lefiVia::xTopOffset(),
                     via->lefiVia::yTopOffset());
     if (via->lefiVia::hasCutPattern())
        fprintf(fout, "    PATTERN %s ;\n", via->lefiVia::cutPattern());
  }
  fprintf(fout, "END %s\n", via->lefiVia::name());

  return;
}
```

```
void lefSpacing(lefiSpacing* spacing) {
  fprintf(fout, "  SAMENET %s %s %g ", spacing->lefiSpacing::name1(),
          spacing->lefiSpacing::name2(), spacing->lefiSpacing::distance());
  if (spacing->lefiSpacing::hasStack())
     fprintf(fout, "STACK ");
  fprintf(fout,";\n");
  return;
}

void lefViaRuleLayer(lefiViaRuleLayer* vLayer) {
  fprintf(fout, "  LAYER %s ;\n", vLayer->lefiViaRuleLayer::name());
  if (vLayer->lefiViaRuleLayer::hasDirection()) {
     if (vLayer->lefiViaRuleLayer::isHorizontal())
        fprintf(fout, "    DIRECTION HORIZONTAL ;\n");
     if (vLayer->lefiViaRuleLayer::isVertical())
        fprintf(fout, "    DIRECTION VERTICAL ;\n");
  }
  if (vLayer->lefiViaRuleLayer::hasEnclosure()) {
     fprintf(fout, "    ENCLOSURE %g %g ;\n",
             vLayer->lefiViaRuleLayer::enclosureOverhang1(),
             vLayer->lefiViaRuleLayer::enclosureOverhang2());
  }
  if (vLayer->lefiViaRuleLayer::hasWidth())
     fprintf(fout, "    WIDTH %g TO %g ;\n",
             vLayer->lefiViaRuleLayer::widthMin(),
             vLayer->lefiViaRuleLayer::widthMax());
  if (vLayer->lefiViaRuleLayer::hasResistance())
     fprintf(fout, "    RESISTANCE %g ;\n",
             vLayer->lefiViaRuleLayer::resistance());
  if (vLayer->lefiViaRuleLayer::hasOverhang())
     fprintf(fout, "    OVERHANG %g ;\n",
             vLayer->lefiViaRuleLayer::overhang());
  if (vLayer->lefiViaRuleLayer::hasMetalOverhang())
     fprintf(fout, "    METALOVERHANG %g ;\n",
             vLayer->lefiViaRuleLayer::metalOverhang());
  if (vLayer->lefiViaRuleLayer::hasSpacing())
     fprintf(fout, "    SPACING %g BY %g ;\n",
             vLayer->lefiViaRuleLayer::spacingStepX(),
             vLayer->lefiViaRuleLayer::spacingStepY());
  if (vLayer->lefiViaRuleLayer::hasRect())
```

```
    fprintf(fout, "    RECT ( %f %f ) ( %f %f ) ;\n",
            vLayer->lefiViaRuleLayer::xl(), vLayer->lefiViaRuleLayer::yl(),
            vLayer->lefiViaRuleLayer::xh(), vLayer->lefiViaRuleLayer::yh());
  return;
}

void prtGeometry(lefiGeometries* geometry) {
  int                  numItems = geometry->lefiGeometries::numItems();
  int                  i, j;
  lefiGeomPath*        path;
  lefiGeomPathIter*    pathIter;
  lefiGeomRect*        rect;
  lefiGeomRectIter*    rectIter;
  lefiGeomPolygon*     polygon;
  lefiGeomPolygonIter* polygonIter;
  lefiGeomVia*         via;
  lefiGeomViaIter*     viaIter;

  for (i = 0; i < numItems; i++) {
     switch (geometry->lefiGeometries::itemType(i)) {
        case  lefiGeomClassE:
             fprintf(fout, "CLASS %s ",
                     geometry->lefiGeometries::getClass(i));
             break;
        case lefiGeomLayerE:
             fprintf(fout, "      LAYER %s ;\n",
                     geometry->lefiGeometries::getLayer(i));
             break;
        case lefiGeomLayerExceptPgNetE:
             fprintf(fout, "      EXCEPTPGNET ;\n");
             break;
        case lefiGeomLayerMinSpacingE:
             fprintf(fout, "      SPACING %g ;\n",
                     geometry->lefiGeometries::getLayerMinSpacing(i));
             break;
        case lefiGeomLayerRuleWidthE:
             fprintf(fout, "      DESIGNRULEWIDTH %g ;\n",
                     geometry->lefiGeometries::getLayerRuleWidth(i));
             break;
        case lefiGeomWidthE:
             fprintf(fout, "      WIDTH %g ;\n",
```

```
                geometry->lefiGeometries::getWidth(i));
            break;
        case lefiGeomPathE:
            path = geometry->lefiGeometries::getPath(i);
            fprintf(fout, "      PATH ");
            for (j = 0; j < path->numPoints; j++) {
               if (j+1 == path->numPoints) // last one on the list
                  fprintf(fout, "      ( %g %g ) ;\n", path->x[j], path->y[j]);
               else
                  fprintf(fout, "      ( %g %g )\n", path->x[j], path->y[j]);
            }
            break;
        case lefiGeomPathIterE:
            pathIter = geometry->lefiGeometries::getPathIter(i);
            fprintf(fout, "      PATH ITERATED ");
            for (j = 0; j < pathIter->numPoints; j++)
               fprintf(fout, "      ( %g %g )\n", pathIter->x[j],
                       pathIter->y[j]);
            fprintf(fout, "      DO %g BY %g STEP %g %g ;\n", pathIter->xStart,
                    pathIter->yStart, pathIter->xStep, pathIter->yStep);
            break;
        case lefiGeomRectE:
            rect = geometry->lefiGeometries::getRect(i);
            fprintf(fout, "      RECT ( %f %f ) ( %f %f ) ;\n", rect->xl,
                    rect->yl, rect->xh, rect->yh);
            break;
        case lefiGeomRectIterE:
            rectIter = geometry->lefiGeometries::getRectIter(i);
            fprintf(fout, "      RECT ITERATE ( %f %f ) ( %f %f )\n",
                    rectIter->xl, rectIter->yl, rectIter->xh, rectIter->yh);
            fprintf(fout, "      DO %g BY %g STEP %g %g ;\n",
                    rectIter->xStart, rectIter->yStart, rectIter->xStep,
                    rectIter->yStep);
            break;
        case lefiGeomPolygonE:
            polygon = geometry->lefiGeometries::getPolygon(i);
            fprintf(fout, "      POLYGON ");
            for (j = 0; j < polygon->numPoints; j++) {
               if (j+1 == polygon->numPoints) // last one on the list
                  fprintf(fout, "      ( %g %g ) ;\n", polygon->x[j],
                          polygon->y[j]);
```

```
                else
                    fprintf(fout, "      ( %g %g )\n", polygon->x[j],
                            polygon->y[j]);
            }
            break;
        case lefiGeomPolygonIterE:
            polygonIter = geometry->lefiGeometries::getPolygonIter(i);
            fprintf(fout, "      POLYGON ITERATE");
            for (j = 0; j < polygonIter->numPoints; j++)
                fprintf(fout, "      ( %g %g )\n", polygonIter->x[j],
                        polygonIter->y[j]);
            fprintf(fout, "      DO %g BY %g STEP %g %g ;\n",
                    polygonIter->xStart, polygonIter->yStart,
                    polygonIter->xStep, polygonIter->yStep);
            break;
        case lefiGeomViaE:
            via = geometry->lefiGeometries::getVia(i);
            fprintf(fout, "      VIA ( %g %g ) %s ;\n", via->x, via->y,
                    via->name);
            break;
        case lefiGeomViaIterE:
            viaIter = geometry->lefiGeometries::getViaIter(i);
            fprintf(fout, "      VIA ITERATE ( %g %g ) %s\n", viaIter->x,
                    viaIter->y, viaIter->name);
            fprintf(fout, "      DO %g BY %g STEP %g %g ;\n",
                    viaIter->xStart, viaIter->yStart,
                    viaIter->xStep, viaIter->yStep);
            break;
        default:
            fprintf(fout, "BOGUS geometries type.\n");
            break;
     }
  }
}

int antennaCB(lefrCallbackType_e c, double value, lefiUserData ud) {
  checkType(c);

  switch (c) {
        case lefrAntennaInputCbkType:
            fprintf(fout, "ANTENNAINPUTGATEAREA %g ;\n", value);
```

```
            break;
        case lefrAntennaInoutCbkType:
            fprintf(fout, "ANTENNAINOUTDIFFAREA %g ;\n", value);
            break;
        case lefrAntennaOutputCbkType:
            fprintf(fout, "ANTENNAOUTPUTDIFFAREA %g ;\n", value);
            break;
        case lefrInputAntennaCbkType:
            fprintf(fout, "INPUTPINANTENNASIZE %g ;\n", value);
            break;
        case lefrOutputAntennaCbkType:
            fprintf(fout, "OUTPUTPINANTENNASIZE %g ;\n", value);
            break;
        case lefrInoutAntennaCbkType:
            fprintf(fout, "INOUTPINANTENNASIZE %g ;\n", value);
            break;
        default:
            fprintf(fout, "BOGUS antenna type.\n");
            break;
  }
  return 0;
}

int arrayBeginCB(lefrCallbackType_e c, const char* name, lefiUserData ud) {
  int  status;

  checkType(c);
  status = lefwStartArray(name);
  if (status != LEFW_OK)
     return status;
  return 0;
}

int arrayCB(lefrCallbackType_e c, lefiArray* a, lefiUserData ud) {
  int              status, i, j, defCaps;
  lefiSitePattern* pattern;
  lefiTrackPattern* track;
  lefiGcellPattern* gcell;

  checkType(c);
  if (a->lefiArray::numSitePattern() > 0) {
```

```
      for (i = 0; i < a->lefiArray::numSitePattern(); i++) {
         pattern = a->lefiArray::sitePattern(i);
         status = lefwArraySite(pattern->lefiSitePattern::name(),
                                pattern->lefiSitePattern::x(),
                                pattern->lefiSitePattern::y(),
                                pattern->lefiSitePattern::orient(),
                                pattern->lefiSitePattern::xStart(),
                                pattern->lefiSitePattern::yStart(),
                                pattern->lefiSitePattern::xStep(),
                                pattern->lefiSitePattern::yStep());
         if (status != LEFW_OK)
            dataError();
      }
   }
   if (a->lefiArray::numCanPlace() > 0) {
      for (i = 0; i < a->lefiArray::numCanPlace(); i++) {
         pattern = a->lefiArray::canPlace(i);
         status = lefwArrayCanplace(pattern->lefiSitePattern::name(),
                                    pattern->lefiSitePattern::x(),
                                    pattern->lefiSitePattern::y(),
                                    pattern->lefiSitePattern::orient(),
                                    pattern->lefiSitePattern::xStart(),
                                    pattern->lefiSitePattern::yStart(),
                                    pattern->lefiSitePattern::xStep(),
                                    pattern->lefiSitePattern::yStep());
         if (status != LEFW_OK)
            dataError();
      }
   }
   if (a->lefiArray::numCannotOccupy() > 0) {
      for (i = 0; i < a->lefiArray::numCannotOccupy(); i++) {
         pattern = a->lefiArray::cannotOccupy(i);
         status = lefwArrayCannotoccupy(pattern->lefiSitePattern::name(),
                                        pattern->lefiSitePattern::x(),
                                        pattern->lefiSitePattern::y(),
                                        pattern->lefiSitePattern::orient(),
                                        pattern->lefiSitePattern::xStart(),
                                        pattern->lefiSitePattern::yStart(),
                                        pattern->lefiSitePattern::xStep(),
                                        pattern->lefiSitePattern::yStep());
         if (status != LEFW_OK)
```

```
         dataError();
      }
   }

   if (a->lefiArray::numTrack() > 0) {
      for (i = 0; i < a->lefiArray::numTrack(); i++) {
         track = a->lefiArray::track(i);
         fprintf(fout, "  TRACKS %s, %g DO %d STEP %g\n",
                 track->lefiTrackPattern::name(),
                 track->lefiTrackPattern::start(),
                 track->lefiTrackPattern::numTracks(),
                 track->lefiTrackPattern::space());
         if (track->lefiTrackPattern::numLayers() > 0) {
            fprintf(fout, "  LAYER ");
            for (j = 0; j < track->lefiTrackPattern::numLayers(); j++)
               fprintf(fout, "%s ", track->lefiTrackPattern::layerName(j));
            fprintf(fout, ";\n");
         }
      }
   }

   if (a->lefiArray::numGcell() > 0) {
      for (i = 0; i < a->lefiArray::numGcell(); i++) {
         gcell = a->lefiArray::gcell(i);
         fprintf(fout, "  GCELLGRID %s, %g DO %d STEP %g\n",
                 gcell->lefiGcellPattern::name(),
                 gcell->lefiGcellPattern::start(),
                 gcell->lefiGcellPattern::numCRs(),
                 gcell->lefiGcellPattern::space());
      }
   }

   if (a->lefiArray::numFloorPlans() > 0) {
      for (i = 0; i < a->lefiArray::numFloorPlans(); i++) {
         status = lefwStartArrayFloorplan(a->lefiArray::floorPlanName(i));
         if (status != LEFW_OK)
            dataError();
         for (j = 0; j < a->lefiArray::numSites(i); j++) {
            pattern = a->lefiArray::site(i, j);
            status = lefwArrayFloorplan(a->lefiArray::siteType(i, j),
                                        pattern->lefiSitePattern::name(),
```

```
                                    pattern->lefiSitePattern::x(),
                                    pattern->lefiSitePattern::y(),
                                    pattern->lefiSitePattern::orient(),
                                    (int)pattern->lefiSitePattern::xStart(),
                                    (int)pattern->lefiSitePattern::yStart(),
                                    pattern->lefiSitePattern::xStep(),
                                    pattern->lefiSitePattern::yStep());
          if (status != LEFW_OK)
             dataError();
       }
    status = lefwEndArrayFloorplan(a->lefiArray::floorPlanName(i));
    if (status != LEFW_OK)
       dataError();
    }
  }

  defCaps = a->lefiArray::numDefaultCaps();
  if (defCaps > 0) {
     status = lefwStartArrayDefaultCap(defCaps);
     if (status != LEFW_OK)
        dataError();
     for (i = 0; i < defCaps; i++) {
        status = lefwArrayDefaultCap(a->lefiArray::defaultCapMinPins(i),
                                     a->lefiArray::defaultCap(i));
        if (status != LEFW_OK)
           dataError();
     }
     status = lefwEndArrayDefaultCap();
     if (status != LEFW_OK)
        dataError();
  }
  return 0;
}

int arrayEndCB(lefrCallbackType_e c, const char* name, lefiUserData ud) {
  int  status;

  checkType(c);
  status = lefwEndArray(name);
  if (status != LEFW_OK)
     return status;
```

```
  return 0;
}

int busBitCharsCB(lefrCallbackType_e c, const char* busBit, lefiUserData ud)
{
  int status;

  checkType(c);
  status = lefwBusBitChars(busBit);
  if (status != LEFW_OK)
     dataError();
  return 0;
}

int caseSensCB(lefrCallbackType_e c, int caseSense, lefiUserData ud) {
  checkType(c);

  if (caseSense == TRUE)
     fprintf(fout, "NAMESCASESENSITIVE ON ;\n");
  else
     fprintf(fout, "NAMESCASESENSITIVE OFF ;\n");
  return 0;
}

int clearanceCB(lefrCallbackType_e c, const char* name, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "CLEARANCEMEASURE %s ;\n", name);
  return 0;
}

int dividerCB(lefrCallbackType_e c, const char* name, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "DIVIDER %s ;\n", name);
  return 0;
}

int noWireExtCB(lefrCallbackType_e c, const char* name, lefiUserData ud) {
  checkType(c);
```

```
  fprintf(fout, "NOWIREEXTENSION %s ;\n", name);
  return 0;
}

int edge1CB(lefrCallbackType_e c, double name, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "EDGERATETHRESHOLD1 %g ;\n", name);
  return 0;
}

int edge2CB(lefrCallbackType_e c, double name, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "EDGERATETHRESHOLD2 %g ;\n", name);
  return 0;
}

int edgeScaleCB(lefrCallbackType_e c, double name, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "EDGERATESCALEFACTORE %g ;\n", name);
  return 0;
}

int dielectricCB(lefrCallbackType_e c, double dielectric, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "DIELECTRIC %g ;\n", dielectric);
  return 0;
}

int irdropBeginCB(lefrCallbackType_e c, void* ptr, lefiUserData ud){
  checkType(c);

  fprintf(fout, "IRDROP\n");
  return 0;
}

int irdropCB(lefrCallbackType_e c, lefiIRDrop* irdrop, lefiUserData ud) {
  int i;
```

```
  checkType(c);

  fprintf(fout, "  TABLE %s ", irdrop->lefiIRDrop::name());
  for (i = 0; i < irdrop->lefiIRDrop::numValues(); i++)
     fprintf(fout, "%g %g ", irdrop->lefiIRDrop::value1(i),
             irdrop->lefiIRDrop::value2(i));
  fprintf(fout, ";\n");
  return 0;
}

int irdropEndCB(lefrCallbackType_e c, void* ptr, lefiUserData ud){
  checkType(c);

  fprintf(fout, "END IRDROP\n");
  return 0;
}

int layerCB(lefrCallbackType_e c, lefiLayer* layer, lefiUserData ud) {
  int i, j, k;
  int numPoints, propNum;
  double *widths, *current;
  lefiLayerDensity* density;
  lefiAntennaPWL* pwl;
  lefiSpacingTable* spTable;
  lefiInfluence* influence;
  lefiParallel* parallel;
  lefiTwoWidths* twoWidths;
  char pType;
  int numMinCut, numMinenclosed;
  lefiAntennaModel* aModel;
  lefiOrthogonal*   ortho;

  checkType(c);
  lefrSetCaseSensitivity(0);

  if (parse65nm)
     layer->lefiLayer::parse65nmRules();

  fprintf(fout, "LAYER %s\n", layer->lefiLayer::name());
  if (layer->lefiLayer::hasType())
     fprintf(fout, "  TYPE %s ;\n", layer->lefiLayer::type());
```

```
if (layer->lefiLayer::hasPitch())
   fprintf(fout, "  PITCH %g ;\n", layer->lefiLayer::pitch());
else if (layer->lefiLayer::hasXYPitch())
   fprintf(fout, "  PITCH %g %g ;\n", layer->lefiLayer::pitchX(),
           layer->lefiLayer::pitchY());
if (layer->lefiLayer::hasOffset())
   fprintf(fout, "  OFFSET %g ;\n", layer->lefiLayer::offset());
else if (layer->lefiLayer::hasXYOffset())
   fprintf(fout, "  OFFSET %g %g ;\n", layer->lefiLayer::offsetX(),
           layer->lefiLayer::offsetY());
if (layer->lefiLayer::hasDiagPitch())
   fprintf(fout, "  DIAGPITCH %g ;\n", layer->lefiLayer::diagPitch());
else if (layer->lefiLayer::hasXYDiagPitch())
   fprintf(fout, "  DIAGPITCH %g %g ;\n", layer->lefiLayer::diagPitchX(),
           layer->lefiLayer::diagPitchY());
if (layer->lefiLayer::hasDiagWidth())
   fprintf(fout, "  DIAGWIDTH %g ;\n", layer->lefiLayer::diagWidth());
if (layer->lefiLayer::hasDiagSpacing())
   fprintf(fout, "  DIAGSPACING %g ;\n", layer->lefiLayer::diagSpacing());
if (layer->lefiLayer::hasWidth())
   fprintf(fout, "  WIDTH %g ;\n", layer->lefiLayer::width());
if (layer->lefiLayer::hasArea())
   fprintf(fout, "  AREA %g ;\n", layer->lefiLayer::area());
if (layer->lefiLayer::hasSlotWireWidth())
   fprintf(fout, "  SLOTWIREWIDTH %g ;\n", layer->lefiLayer::slotWireWidth());
if (layer->lefiLayer::hasSlotWireLength())
   fprintf(fout, "  SLOTWIRELENGTH %g ;\n",
           layer->lefiLayer::slotWireLength());
if (layer->lefiLayer::hasSlotWidth())
   fprintf(fout, "  SLOTWIDTH %g ;\n", layer->lefiLayer::slotWidth());
if (layer->lefiLayer::hasSlotLength())
   fprintf(fout, "  SLOTLENGTH %g ;\n", layer->lefiLayer::slotLength());
if (layer->lefiLayer::hasMaxAdjacentSlotSpacing())
   fprintf(fout, "  MAXADJACENTSLOTSPACING %g ;\n",
           layer->lefiLayer::maxAdjacentSlotSpacing());
if (layer->lefiLayer::hasMaxCoaxialSlotSpacing())
   fprintf(fout, "  MAXCOAXIALSLOTSPACING %g ;\n",
           layer->lefiLayer::maxCoaxialSlotSpacing());
if (layer->lefiLayer::hasMaxEdgeSlotSpacing())
   fprintf(fout, "  MAXEDGESLOTSPACING %g ;\n",
           layer->lefiLayer::maxEdgeSlotSpacing());
```

```
if (layer->lefiLayer::hasMaxFloatingArea())          // 5.7
   fprintf(fout, "  MAXFLOATINGAREA %g ;\n",
           layer->lefiLayer::maxFloatingArea());
if (layer->lefiLayer::hasArraySpacing()) {           // 5.7
   fprintf(fout, "  ARRAYSPACING ");
   if (layer->lefiLayer::hasLongArray())
      fprintf(fout, "LONGARRAY ");
   if (layer->lefiLayer::hasViaWidth())
      fprintf(fout, "WIDTH %g ", layer->lefiLayer::viaWidth());
   fprintf(fout, "CUTSPACING %g", layer->lefiLayer::cutSpacing());
   for (i = 0; i < layer->lefiLayer::numArrayCuts(); i++)
      fprintf(fout, "\n\tARRAYCUTS %g SPACING %g",
              layer->lefiLayer::arrayCuts(i),
              layer->lefiLayer::arraySpacing(i));
   fprintf(fout, " ;\n");
}
if (layer->lefiLayer::hasSplitWireWidth())
   fprintf(fout, "  SPLITWIREWIDTH %g ;\n",
           layer->lefiLayer::splitWireWidth());
if (layer->lefiLayer::hasMinimumDensity())
   fprintf(fout, "  MINIMUMDENSITY %g ;\n",
           layer->lefiLayer::minimumDensity());
if (layer->lefiLayer::hasMaximumDensity())
   fprintf(fout, "  MAXIMUMDENSITY %g ;\n",
           layer->lefiLayer::maximumDensity());
if (layer->lefiLayer::hasDensityCheckWindow())
   fprintf(fout, "  DENSITYCHECKWINDOW %g %g ;\n",
           layer->lefiLayer::densityCheckWindowLength(),
           layer->lefiLayer::densityCheckWindowWidth());
if (layer->lefiLayer::hasDensityCheckStep())
   fprintf(fout, "  DENSITYCHECKSTEP %g ;\n",
           layer->lefiLayer::densityCheckStep());
if (layer->lefiLayer::hasFillActiveSpacing())
   fprintf(fout, "  FILLACTIVESPACING %g ;\n",
           layer->lefiLayer::fillActiveSpacing());
// 5.4.1
numMinCut = layer->lefiLayer::numMinimumcut();
if (numMinCut > 0) {
   for (i = 0; i < numMinCut; i++) {
       fprintf(fout, "  MINIMUMCUT %d WIDTH %g ",
            layer->lefiLayer::minimumcut(i),
```

```
                layer->lefiLayer::minimumcutWidth(i));
            if (layer->lefiLayer::hasMinimumcutWithin(i))
               fprintf(fout, "WITHIN %g ", layer->lefiLayer::minimumcutWithin(i));
            if (layer->lefiLayer::hasMinimumcutConnection(i))
               fprintf(fout, "%s ", layer->lefiLayer::minimumcutConnection(i));
            if (layer->lefiLayer::hasMinimumcutNumCuts(i))
               fprintf(fout, "LENGTH %g WITHIN %g ",
               layer->lefiLayer::minimumcutLength(i),
               layer->lefiLayer::minimumcutDistance(i));
            fprintf(fout, ";\n");
        }
    }
    // 5.4.1
    if (layer->lefiLayer::hasMaxwidth()) {
        fprintf(fout, "  MAXWIDTH %g ;\n", layer->lefiLayer::maxwidth());
    }
    // 5.5
    if (layer->lefiLayer::hasMinwidth()) {
        fprintf(fout, "  MINWIDTH %g ;\n", layer->lefiLayer::minwidth());
    }
    // 5.5
    numMinenclosed = layer->lefiLayer::numMinenclosedarea();
    if (numMinenclosed > 0) {
        for (i = 0; i < numMinenclosed; i++) {
            fprintf(fout, "  MINENCLOSEDAREA %g ",
                layer->lefiLayer::minenclosedarea(i));
            if (layer->lefiLayer::hasMinenclosedareaWidth(i))
                fprintf(fout, "MINENCLOSEDAREAWIDTH %g ",
                        layer->lefiLayer::minenclosedareaWidth(i));
            fprintf (fout, ";\n");
        }
    }
    // 5.4.1 & 5.6
    if (layer->lefiLayer::hasMinstep()) {
        for (i = 0; i < layer->lefiLayer::numMinstep(); i++) {
           fprintf(fout, "  MINSTEP %g ", layer->lefiLayer::minstep(i));
           if (layer->lefiLayer::hasMinstepType(i))
              fprintf(fout, "%s ", layer->lefiLayer::minstepType(i));
           if (layer->lefiLayer::hasMinstepLengthsum(i))
              fprintf(fout, "LENGTHSUM %g ",
                      layer->lefiLayer::minstepLengthsum(i));
```

```
      if (layer->lefiLayer::hasMinstepMaxedges(i))
         fprintf(fout, "MAXEDGES %d ", layer->lefiLayer::minstepMaxedges(i));
      fprintf(fout, ";\n");
   }
}
// 5.4.1
if (layer->lefiLayer::hasProtrusion()) {
   fprintf(fout, "  PROTRUSIONWIDTH %g LENGTH %g WIDTH %g ;\n",
           layer->lefiLayer::protrusionWidth1(),
           layer->lefiLayer::protrusionLength(),
           layer->lefiLayer::protrusionWidth2());
}
if (layer->lefiLayer::hasSpacingNumber()) {
   for (i = 0; i < layer->lefiLayer::numSpacing(); i++) {
     fprintf(fout, "  SPACING %g ", layer->lefiLayer::spacing(i));
     if (layer->lefiLayer::hasSpacingName(i))
        fprintf(fout, "LAYER %s ", layer->lefiLayer::spacingName(i));
     if (layer->lefiLayer::hasSpacingLayerStack(i))
        fprintf(fout, "STACK ");                          // 5.7
     if (layer->lefiLayer::hasSpacingAdjacent(i))
        fprintf(fout, "ADJACENTCUTS %d WITHIN %g ",
                layer->lefiLayer::spacingAdjacentCuts(i),
                layer->lefiLayer::spacingAdjacentWithin(i));
     if (layer->lefiLayer::hasSpacingAdjacentExcept(i))    // 5.7
        fprintf(fout, "EXCEPTSAMEPGNET ");
     if (layer->lefiLayer::hasSpacingCenterToCenter(i))
        fprintf(fout, "CENTERTOCENTER ");
     if (layer->lefiLayer::hasSpacingSamenet(i))           // 5.7
        fprintf(fout, "SAMENET ");
         if (layer->lefiLayer::hasSpacingSamenetPGonly(i)) // 5.7
            fprintf(fout, "PGONLY ");
     if (layer->lefiLayer::hasSpacingArea(i))              // 5.7
        fprintf(fout, "AREA %g ", layer->lefiLayer::spacingArea(i));
     if (layer->lefiLayer::hasSpacingRange(i)) {
        fprintf(fout, "RANGE %g %g ", layer->lefiLayer::spacingRangeMin(i),
                layer->lefiLayer::spacingRangeMax(i));
        if (layer->lefiLayer::hasSpacingRangeUseLengthThreshold(i))
           fprintf(fout, "USELENGTHTHRESHOLD ");
        else if (layer->lefiLayer::hasSpacingRangeInfluence(i)) {
            fprintf(fout, "INFLUENCE %g ",
               layer->lefiLayer::spacingRangeInfluence(i));
```

```
          if (layer->lefiLayer::hasSpacingRangeInfluenceRange(i))
             fprintf(fout, "RANGE %g %g ",
                layer->lefiLayer::spacingRangeInfluenceMin(i),
                layer->lefiLayer::spacingRangeInfluenceMax(i));
       } else if (layer->lefiLayer::hasSpacingRangeRange(i))
           fprintf(fout, "RANGE %g %g ",
             layer->lefiLayer::spacingRangeRangeMin(i),
             layer->lefiLayer::spacingRangeRangeMax(i));
   } else if (layer->lefiLayer::hasSpacingLengthThreshold(i)) {
       fprintf(fout, "LENGTHTHRESHOLD %g ",
          layer->lefiLayer::spacingLengthThreshold(i));
       if (layer->lefiLayer::hasSpacingLengthThresholdRange(i))
          fprintf(fout, "RANGE %g %g",
             layer->lefiLayer::spacingLengthThresholdRangeMin(i),
             layer->lefiLayer::spacingLengthThresholdRangeMax(i));
   } else if (layer->lefiLayer::hasSpacingNotchLength(i)) {// 5.7
       fprintf(fout, "NOTCHLENGTH %g",
              layer->lefiLayer::spacingNotchLength(i));
   } else if (layer->lefiLayer::hasSpacingEndOfNotchWidth(i)) // 5.7
       fprintf(fout, "ENDOFNOTCHWIDTH %g NOTCHSPACING %g, NOTCHLENGTH %g",
              layer->lefiLayer::spacingEndOfNotchWidth(i),
              layer->lefiLayer::spacingEndOfNotchSpacing(i),
              layer->lefiLayer::spacingEndOfNotchLength(i));

   if (layer->lefiLayer::hasSpacingParallelOverlap(i))   // 5.7
      fprintf(fout, "PARALLELOVERLAP ");
   if (layer->lefiLayer::hasSpacingEndOfLine(i)) {       // 5.7
      fprintf(fout, "ENDOFLINE %g WITHING %g ",
         layer->lefiLayer::spacingEolWidth(i),
         layer->lefiLayer::spacingEolWithin(i));
      if (layer->lefiLayer::hasSpacingParellelEdge(i)) {
         fprintf(fout, "PARALLELEDGE %g WITHING %g ",
            layer->lefiLayer::spacingParSpace(i),
            layer->lefiLayer::spacingParWithin(i));
         if (layer->lefiLayer::hasSpacingTwoEdges(i)) {
            fprintf(fout, "TWOEDGES ");
         }
      }
   }
   fprintf(fout, ";\n");
}
```

```
}
if (layer->lefiLayer::hasSpacingTableOrtho()) {            // 5.7
   fprintf(fout, "SPACINGTABLE ORTHOGONAL");
   ortho = layer->lefiLayer::orthogonal();
   for (i = 0; i < ortho->lefiOrthogonal::numOrthogonal(); i++) {
      fprintf(fout, "\n   WITHIN %g SPACING %g",
              ortho->lefiOrthogonal::cutWithin(i),
              ortho->lefiOrthogonal::orthoSpacing(i));
   }
   fprintf(fout, ";\n");
}
for (i = 0; i < layer->lefiLayer::numEnclosure(); i++) {
   fprintf(fout, "ENCLOSURE ");
   if (layer->lefiLayer::hasEnclosureRule(i))
      fprintf(fout, "%s ", layer->lefiLayer::enclosureRule(i));
   fprintf(fout, "%g %g ", layer->lefiLayer::enclosureOverhang1(i),
                           layer->lefiLayer::enclosureOverhang2(i));
   if (layer->lefiLayer::hasEnclosureWidth(i))
      fprintf(fout, "WIDTH %g ", layer->lefiLayer::enclosureMinWidth(i));
   if (layer->lefiLayer::hasEnclosureExceptExtraCut(i))
      fprintf(fout, "EXCEPTEXTRACUT %g ",
              layer->lefiLayer::enclosureExceptExtraCut(i));
   if (layer->lefiLayer::hasEnclosureMinLength(i))
      fprintf(fout, "LENGTH %g ", layer->lefiLayer::enclosureMinLength(i));
   fprintf(fout, ";\n");
}
for (i = 0; i < layer->lefiLayer::numPreferEnclosure(); i++) {
   fprintf(fout, "PREFERENCLOSURE ");
   if (layer->lefiLayer::hasPreferEnclosureRule(i))
      fprintf(fout, "%s ", layer->lefiLayer::preferEnclosureRule(i));
   fprintf(fout, "%g %g ", layer->lefiLayer::preferEnclosureOverhang1(i),
                           layer->lefiLayer::preferEnclosureOverhang2(i));
   if (layer->lefiLayer::hasPreferEnclosureWidth(i))
      fprintf(fout, "WIDTH %g ",layer->lefiLayer::preferEnclosureMinWidth(i));
   fprintf(fout, ";\n");
}
if (layer->lefiLayer::hasResistancePerCut())
   fprintf(fout, "  RESISTANCE %g ;\n",
           layer->lefiLayer::resistancePerCut());
if (layer->lefiLayer::hasCurrentDensityPoint())
   fprintf(fout, "  CURRENTDEN %g ;\n",
```

```
            layer->lefiLayer::currentDensityPoint());
  if (layer->lefiLayer::hasCurrentDensityArray()) {
     layer->lefiLayer::currentDensityArray(&numPoints, &widths, &current);
     for (i = 0; i < numPoints; i++)
         fprintf(fout, "  CURRENTDEN ( %g %g ) ;\n", widths[i], current[i]);
  }
  if (layer->lefiLayer::hasDirection())
     fprintf(fout, "  DIRECTION %s ;\n", layer->lefiLayer::direction());
  if (layer->lefiLayer::hasResistance())
     fprintf(fout, "  RESISTANCE RPERSQ %g ;\n",
             layer->lefiLayer::resistance());
  if (layer->lefiLayer::hasCapacitance())
     fprintf(fout, "  CAPACITANCE CPERSQDIST %g ;\n",
             layer->lefiLayer::capacitance());
  if (layer->lefiLayer::hasEdgeCap())
     fprintf(fout, "  EDGECAPACITANCE %g ;\n", layer->lefiLayer::edgeCap());
  if (layer->lefiLayer::hasHeight())
     fprintf(fout, "  TYPE %g ;\n", layer->lefiLayer::height());
  if (layer->lefiLayer::hasThickness())
     fprintf(fout, "  THICKNESS %g ;\n", layer->lefiLayer::thickness());
  if (layer->lefiLayer::hasWireExtension())
     fprintf(fout, "  WIREEXTENSION %g ;\n", layer->lefiLayer::wireExtension());
  if (layer->lefiLayer::hasShrinkage())
     fprintf(fout, "  SHRINKAGE %g ;\n", layer->lefiLayer::shrinkage());
  if (layer->lefiLayer::hasCapMultiplier())
     fprintf(fout, "  CAPMULTIPLIER %g ;\n", layer->lefiLayer::capMultiplier());
  if (layer->lefiLayer::hasAntennaArea())
     fprintf(fout, "  ANTENNAAREAFACTOR %g ;\n",
             layer->lefiLayer::antennaArea());
  if (layer->lefiLayer::hasAntennaLength())
     fprintf(fout, "  ANTENNALENGTHFACTOR %g ;\n",
             layer->lefiLayer::antennaLength());

  // 5.5 AntennaModel
  for (i = 0; i < layer->lefiLayer::numAntennaModel(); i++) {
     aModel = layer->lefiLayer::antennaModel(i);

     fprintf(fout, "  ANTENNAMODEL %s ;\n",
             aModel->lefiAntennaModel::antennaOxide());

     if (aModel->lefiAntennaModel::hasAntennaAreaRatio())
```

```
      fprintf(fout, "  ANTENNAAREARATIO %g ;\n",
              aModel->lefiAntennaModel::antennaAreaRatio());
   if (aModel->lefiAntennaModel::hasAntennaDiffAreaRatio())
      fprintf(fout, "  ANTENNADIFFAREARATIO %g ;\n",
              aModel->lefiAntennaModel::antennaDiffAreaRatio());
   else if (aModel->lefiAntennaModel::hasAntennaDiffAreaRatioPWL()) {
      pwl = aModel->lefiAntennaModel::antennaDiffAreaRatioPWL();
      fprintf(fout, "  ANTENNADIFFAREARATIO PWL ( ");
      for (j = 0; j < pwl->lefiAntennaPWL::numPWL(); j++)
         fprintf(fout, "( %g %g ) ", pwl->lefiAntennaPWL::PWLdiffusion(j),
                 pwl->lefiAntennaPWL::PWLratio(j));
      fprintf(fout, ") ;\n");
   }
   if (aModel->lefiAntennaModel::hasAntennaCumAreaRatio())
      fprintf(fout, "  ANTENNACUMAREARATIO %g ;\n",
              aModel->lefiAntennaModel::antennaCumAreaRatio());
   if (aModel->lefiAntennaModel::hasAntennaCumDiffAreaRatio())
      fprintf(fout, "  ANTENNACUMDIFFAREARATIO %g\n",
              aModel->lefiAntennaModel::antennaCumDiffAreaRatio());
   if (aModel->lefiAntennaModel::hasAntennaCumDiffAreaRatioPWL()) {
      pwl = aModel->lefiAntennaModel::antennaCumDiffAreaRatioPWL();
      fprintf(fout, "  ANTENNACUMDIFFAREARATIO PWL ( ");
      for (j = 0; j < pwl->lefiAntennaPWL::numPWL(); j++)
         fprintf(fout, "( %g %g ) ", pwl->lefiAntennaPWL::PWLdiffusion(j),
                 pwl->lefiAntennaPWL::PWLratio(j));
      fprintf(fout, ") ;\n");
   }
   if (aModel->lefiAntennaModel::hasAntennaAreaFactor()) {
      fprintf(fout, "  ANTENNAAREAFACTOR %g ",
              aModel->lefiAntennaModel::antennaAreaFactor());
      if (aModel->lefiAntennaModel::hasAntennaAreaFactorDUO())
         fprintf(fout, "  DIFFUSEONLY ");
      fprintf(fout, ";\n");
   }
   if (aModel->lefiAntennaModel::hasAntennaSideAreaRatio())
      fprintf(fout, "  ANTENNASIDEAREARATIO %g ;\n",
              aModel->lefiAntennaModel::antennaSideAreaRatio());
   if (aModel->lefiAntennaModel::hasAntennaDiffSideAreaRatio())
      fprintf(fout, "  ANTENNADIFFSIDEAREARATIO %g\n",
              aModel->lefiAntennaModel::antennaDiffSideAreaRatio());
   else if (aModel->lefiAntennaModel::hasAntennaDiffSideAreaRatioPWL()) {
```

```
      pwl = aModel->lefiAntennaModel::antennaDiffSideAreaRatioPWL();
      fprintf(fout, "  ANTENNADIFFSIDEAREARATIO PWL ( ");
      for (j = 0; j < pwl->lefiAntennaPWL::numPWL(); j++)
         fprintf(fout, "( %g %g ) ", pwl->lefiAntennaPWL::PWLdiffusion(j),
                 pwl->lefiAntennaPWL::PWLratio(j));
      fprintf(fout, ") ;\n");
   }
   if (aModel->lefiAntennaModel::hasAntennaCumSideAreaRatio())
      fprintf(fout, "  ANTENNACUMSIDEAREARATIO %g ;\n",
              aModel->lefiAntennaModel::antennaCumSideAreaRatio());
   if (aModel->lefiAntennaModel::hasAntennaCumDiffSideAreaRatio())
      fprintf(fout, "  ANTENNACUMDIFFSIDEAREARATIO %g\n",
              aModel->lefiAntennaModel::antennaCumDiffSideAreaRatio());
   else if (aModel->lefiAntennaModel::hasAntennaCumDiffSideAreaRatioPWL()) {
      pwl = aModel->lefiAntennaModel::antennaCumDiffSideAreaRatioPWL();
      fprintf(fout, "  ANTENNACUMDIFFSIDEAREARATIO PWL ( ");
      for (j = 0; j < pwl->lefiAntennaPWL::numPWL(); j++)
         fprintf(fout, "( %g %g ) ", pwl->lefiAntennaPWL::PWLdiffusion(j),
                 pwl->lefiAntennaPWL::PWLratio(j));
      fprintf(fout, ") ;\n");
   }
   if (aModel->lefiAntennaModel::hasAntennaSideAreaFactor()) {
      fprintf(fout, "  ANTENNASIDEAREAFACTOR %g ",
              aModel->lefiAntennaModel::antennaSideAreaFactor());
      if (aModel->lefiAntennaModel::hasAntennaSideAreaFactorDUO())
         fprintf(fout, "  DIFFUSEONLY ");
      fprintf(fout, ";\n");
   }
   if (aModel->lefiAntennaModel::hasAntennaCumRoutingPlusCut())
      fprintf(fout, "  ANTENNACUMROUTINGPLUSCUT ;\n");
   if (aModel->lefiAntennaModel::hasAntennaGatePlusDiff())
      fprintf(fout, "  ANTENNAGATEPLUSDIFF %g ;\n",
              aModel->lefiAntennaModel::antennaGatePlusDiff());
   if (aModel->lefiAntennaModel::hasAntennaAreaMinusDiff())
      fprintf(fout, "  ANTENNAAREAMINUSDIFF %g ;\n",
              aModel->lefiAntennaModel::antennaAreaMinusDiff());
   if (aModel->lefiAntennaModel::hasAntennaAreaDiffReducePWL()) {
      pwl = aModel->lefiAntennaModel::antennaAreaDiffReducePWL();
      fprintf(fout, "  ANTENNAAREADIFFREDUCEPWL ( ");
      for (j = 0; j < pwl->lefiAntennaPWL::numPWL(); j++)
         fprintf(fout, "( %g %g ) ", pwl->lefiAntennaPWL::PWLdiffusion(j),
```

```
                pwl->lefiAntennaPWL::PWLratio(j));
        fprintf(fout, ") ;\n");
    }
}

if (layer->lefiLayer::numAccurrentDensity()) {
    for (i = 0; i < layer->lefiLayer::numAccurrentDensity(); i++) {
        density = layer->lefiLayer::accurrent(i);
        fprintf(fout, "  ACCURRENTDENSITY %s", density->type());
        if (density->hasOneEntry())
            fprintf(fout, " %g ;\n", density->oneEntry());
        else {
            fprintf(fout, "\n");
            if (density->numFrequency()) {
                fprintf(fout, "    FREQUENCY");
                for (j = 0; j < density->numFrequency(); j++)
                    fprintf(fout, " %g", density->frequency(j));
                fprintf(fout, " ;\n");
            }
            if (density->numCutareas()) {
                fprintf(fout, "    CUTAREA");
                for (j = 0; j < density->numCutareas(); j++)
                    fprintf(fout, " %g", density->cutArea(j));
                fprintf(fout, " ;\n");
            }
            if (density->numWidths()) {
                fprintf(fout, "    WIDTH");
                for (j = 0; j < density->numWidths(); j++)
                    fprintf(fout, " %g", density->width(j));
                fprintf(fout, " ;\n");
            }
            if (density->numTableEntries()) {
                k = 5;
                fprintf(fout, "    TABLEENTRIES");
                for (j = 0; j < density->numTableEntries(); j++)
                    if (k > 4) {
                        fprintf(fout, "\n     %g", density->tableEntry(j));
                        k = 1;
                    } else {
                        fprintf(fout, " %g", density->tableEntry(j));
                        k++;
```

```
                  }
               fprintf(fout, " ;\n");
            }
         }
      }
   }
   if (layer->lefiLayer::numDccurrentDensity()) {
      for (i = 0; i < layer->lefiLayer::numDccurrentDensity(); i++) {
          density = layer->lefiLayer::dccurrent(i);
          fprintf(fout, "  DCCURRENTDENSITY %s", density->type());
          if (density->hasOneEntry())
              fprintf(fout, " %g ;\n", density->oneEntry());
          else {
              fprintf(fout, "\n");
              if (density->numCutareas()) {
                 fprintf(fout, "    CUTAREA");
                 for (j = 0; j < density->numCutareas(); j++)
                    fprintf(fout, " %g", density->cutArea(j));
                 fprintf(fout, " ;\n");
              }
              if (density->numWidths()) {
                 fprintf(fout, "    WIDTH");
                 for (j = 0; j < density->numWidths(); j++)
                    fprintf(fout, " %g", density->width(j));
                 fprintf(fout, " ;\n");
              }
              if (density->numTableEntries()) {
                 fprintf(fout, "    TABLEENTRIES");
                 for (j = 0; j < density->numTableEntries(); j++)
                    fprintf(fout, " %g", density->tableEntry(j));
                 fprintf(fout, " ;\n");
              }
          }
      }
   }

   for (i = 0; i < layer->lefiLayer::numSpacingTable(); i++) {
      spTable = layer->lefiLayer::spacingTable(i);
      fprintf(fout, "   SPACINGTABLE\n");
      if (spTable->lefiSpacingTable::isInfluence()) {
         influence = spTable->lefiSpacingTable::influence();
```

```
      fprintf(fout, "      INFLUENCE");
      for (j = 0; j < influence->lefiInfluence::numInfluenceEntry(); j++) {
         fprintf(fout, "\n          WIDTH %g WITHIN %g SPACING %g",
                 influence->lefiInfluence::width(j),
                 influence->lefiInfluence::distance(j),
                 influence->lefiInfluence::spacing(j));
      }
      fprintf(fout, " ;\n");
   } else if (spTable->lefiSpacingTable::isParallel()){
      parallel = spTable->lefiSpacingTable::parallel();
      fprintf(fout, "      PARALLELRUNLENGTH");
      for (j = 0; j < parallel->lefiParallel::numLength(); j++) {
         fprintf(fout, " %g", parallel->lefiParallel::length(j));
      }
      for (j = 0; j < parallel->lefiParallel::numWidth(); j++) {
         fprintf(fout, "\n          WIDTH %g",
                 parallel->lefiParallel::width(j));
         for (k = 0; k < parallel->lefiParallel::numLength(); k++) {
            fprintf(fout, " %g", parallel->lefiParallel::widthSpacing(j, k));
         }
      }
      fprintf(fout, " ;\n");
   } else {    // 5.7 TWOWIDTHS
      twoWidths = spTable->lefiSpacingTable::twoWidths();
      fprintf(fout, "      TWOWIDTHS");
      for (j = 0; j < twoWidths->lefiTwoWidths::numWidth(); j++) {
         fprintf(fout, "\n          WIDTH %g ",
                 twoWidths->lefiTwoWidths::width(j));
         if (twoWidths->lefiTwoWidths::hasWidthPRL(j))
            fprintf(fout, "PRL %g ", twoWidths->lefiTwoWidths::widthPRL(j));
         for (k = 0; k < twoWidths->lefiTwoWidths::numWidthSpacing(j); k++)
            fprintf(fout, "%g ",twoWidths->lefiTwoWidths::widthSpacing(j, k));
      }
      fprintf(fout, " ;\n");
   }
}

propNum = layer->lefiLayer::numProps();
if (propNum > 0) {
   fprintf(fout, "  PROPERTY ");
   for (i = 0; i < propNum; i++) {
```

```
        // value can either be a string or number
        fprintf(fout, "%s ", layer->lefiLayer::propName(i));
        if (layer->lefiLayer::propIsNumber(i))
            fprintf(fout, "%g ", layer->lefiLayer::propNumber(i));
        if (layer->lefiLayer::propIsString(i))
            fprintf(fout, "%s ", layer->lefiLayer::propValue(i));
        pType = layer->lefiLayer::propType(i);
        switch (pType) {
           case 'R': fprintf(fout, "REAL ");
                     break;
           case 'I': fprintf(fout, "INTEGER ");
                     break;
           case 'S': fprintf(fout, "STRING ");
                     break;
           case 'Q': fprintf(fout, "QUOTESTRING ");
                     break;
           case 'N': fprintf(fout, "NUMBER ");
                     break;
        }
     }
     fprintf(fout, ";\n");
  }
  if (layer->lefiLayer::hasDiagMinEdgeLength())
     fprintf(fout, "  DIAGMINEDGELENGTH %g ;\n",
             layer->lefiLayer::diagMinEdgeLength());
  if (layer->lefiLayer::numMinSize()) {
     fprintf(fout, "  MINSIZE ");
     for (i = 0; i < layer->lefiLayer::numMinSize(); i++) {
        fprintf(fout, "%g %g ", layer->lefiLayer::minSizeWidth(i),
                                layer->lefiLayer::minSizeLength(i));
     }
     fprintf(fout, ";\n");
  }

  fprintf(fout, "END %s\n", layer->lefiLayer::name());

  // Set it to case sensitive from here on
  lefrSetCaseSensitivity(1);

  return 0;
}
```

```
int macroBeginCB(lefrCallbackType_e c, const char* macroName, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "MACRO %s\n",  macroName);
  return 0;
}

int macroClassTypeCB(lefrCallbackType_e c, const char* macroClassType,
                     lefiUserData ud) {
  checkType(c);

  fprintf(fout, "MACRO CLASS %s\n",  macroClassType);
  return 0;
}

int macroCB(lefrCallbackType_e c, lefiMacro* macro, lefiUserData ud) {
  lefiSitePattern* pattern;
  int              propNum, i, hasPrtSym = 0;

  checkType(c);

  if (macro->lefiMacro::hasClass())
     fprintf(fout, "  CLASS %s ;\n", macro->lefiMacro::macroClass());
  if (macro->lefiMacro::hasEEQ())
     fprintf(fout, "  EEQ %s ;\n", macro->lefiMacro::EEQ());
  if (macro->lefiMacro::hasLEQ())
     fprintf(fout, "  LEQ %s ;\n", macro->lefiMacro::LEQ());
  if (macro->lefiMacro::hasSource())
     fprintf(fout, "  SOURCE %s ;\n", macro->lefiMacro::source());
  if (macro->lefiMacro::hasXSymmetry()) {
     fprintf(fout, "  SYMMETRY X ");
     hasPrtSym = 1;
  }
  if (macro->lefiMacro::hasYSymmetry()) {   // print X Y & R90 in one line
     if (!hasPrtSym) {
        fprintf(fout, "  SYMMETRY Y ");
        hasPrtSym = 1;
     }
     else
        fprintf(fout, "Y ");
```

```
}
if (macro->lefiMacro::has90Symmetry()) {
   if (!hasPrtSym) {
      fprintf(fout, "  SYMMETRY R90 ");
      hasPrtSym = 1;
   }
   else
      fprintf(fout, "R90 ");
}
if (hasPrtSym) {
   fprintf (fout, ";\n");
   hasPrtSym = 0;
}
if (macro->lefiMacro::hasSiteName())
   fprintf(fout, "  SITE %s ;\n", macro->lefiMacro::siteName());
if (macro->lefiMacro::hasSitePattern()) {
   for (i = 0; i < macro->lefiMacro::numSitePattern(); i++ ) {
     pattern = macro->lefiMacro::sitePattern(i);
     if (pattern->lefiSitePattern::hasStepPattern()) {
        fprintf(fout, "  SITE %s %g %g %s DO %g BY %g STEP %g %g ;\n",
              pattern->lefiSitePattern::name(), pattern->lefiSitePattern::x(),
              pattern->lefiSitePattern::y(),
              orientStr(pattern->lefiSitePattern::orient()),
              pattern->lefiSitePattern::xStart(),
              pattern->lefiSitePattern::yStart(),
              pattern->lefiSitePattern::xStep(),
              pattern->lefiSitePattern::yStep());
     } else {
        fprintf(fout, "  SITE %s %g %g %s ;\n",
              pattern->lefiSitePattern::name(), pattern->lefiSitePattern::x(),
              pattern->lefiSitePattern::y(),
              orientStr(pattern->lefiSitePattern::orient()));
     }
   }
}
if (macro->lefiMacro::hasSize())
   fprintf(fout, "  SIZE %g BY %g ;\n", macro->lefiMacro::sizeX(),
           macro->lefiMacro::sizeY());

if (macro->lefiMacro::hasForeign()) {
   for (i = 0; i < macro->lefiMacro::numForeigns(); i++) {
```

```
        fprintf(fout, "  FOREIGN %s ", macro->lefiMacro::foreignName(i));
        if (macro->lefiMacro::hasForeignPoint(i)) {
           fprintf(fout, "( %g %g ) ", macro->lefiMacro::foreignX(i),
                   macro->lefiMacro::foreignY(i));
           if (macro->lefiMacro::hasForeignOrient(i))
              fprintf(fout, "%s ", macro->lefiMacro::foreignOrientStr(i));
        }
        fprintf(fout, ";\n");
     }
  }
  if (macro->lefiMacro::hasOrigin())
     fprintf(fout, "  ORIGIN ( %g %g ) ;\n", macro->lefiMacro::originX(),
             macro->lefiMacro::originY());
  if (macro->lefiMacro::hasPower())
     fprintf(fout, "  POWER %g ;\n", macro->lefiMacro::power());
  propNum = macro->lefiMacro::numProperties();
  if (propNum > 0) {
     fprintf(fout, "  PROPERTY ");
     for (i = 0; i < propNum; i++) {
        // value can either be a string or number
        if (macro->lefiMacro::propValue(i)) {
           fprintf(fout, "%s %s ", macro->lefiMacro::propName(i),
                   macro->lefiMacro::propValue(i));
        }
        else
           fprintf(fout, "%s %g ", macro->lefiMacro::propName(i),
                   macro->lefiMacro::propNum(i));

        switch (macro->lefiMacro::propType(i)) {
           case 'R': fprintf(fout, "REAL ");
                     break;
           case 'I': fprintf(fout, "INTEGER ");
                     break;
           case 'S': fprintf(fout, "STRING ");
                     break;
           case 'Q': fprintf(fout, "QUOTESTRING ");
                     break;
           case 'N': fprintf(fout, "NUMBER ");
                     break;
        }
     }
```

```
    fprintf(fout, ";\n");
  }
  return 0;
}

int macroEndCB(lefrCallbackType_e c, const char* macroName, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "END %s\n", macroName);
  return 0;
}

int manufacturingCB(lefrCallbackType_e c, double num, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "MANUFACTURINGGRID %g ;\n", num);
  return 0;
}

int maxStackViaCB(lefrCallbackType_e c, lefiMaxStackVia* maxStack,
  lefiUserData ud) {
  checkType(c);

  fprintf(fout, "MAXVIASTACK %d ", maxStack->lefiMaxStackVia::maxStackVia());
  if (maxStack->lefiMaxStackVia::hasMaxStackViaRange())
     fprintf(fout, "RANGE %s %s ",
             maxStack->lefiMaxStackVia::maxStackViaBottomLayer(),
             maxStack->lefiMaxStackVia::maxStackViaTopLayer());
  fprintf(fout, ";\n");
  return 0;
}

int minFeatureCB(lefrCallbackType_e c, lefiMinFeature* min, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "MINFEATURE %g %g ;\n", min->lefiMinFeature::one(),
          min->lefiMinFeature::two());
  return 0;
}

int nonDefaultCB(lefrCallbackType_e c, lefiNonDefault* def, lefiUserData ud) {
```

```
int          i;
lefiVia*     via;
lefiSpacing* spacing;

checkType(c);

fprintf(fout, "NONDEFAULTRULE %s\n", def->lefiNonDefault::name());
if (def->lefiNonDefault::hasHardspacing())
   fprintf(fout, "  HARDSPACING ;\n");
for (i = 0; i < def->lefiNonDefault::numLayers(); i++) {
   fprintf(fout, "  LAYER %s\n", def->lefiNonDefault::layerName(i));
   if (def->lefiNonDefault::hasLayerWidth(i))
      fprintf(fout, "    WIDTH %g ;\n", def->lefiNonDefault::layerWidth(i));
   if (def->lefiNonDefault::hasLayerSpacing(i))
      fprintf(fout, "    SPACING %g ;\n",
              def->lefiNonDefault::layerSpacing(i));
   if (def->lefiNonDefault::hasLayerDiagWidth(i))
      fprintf(fout, "    DIAGWIDTH %g ;\n",
              def->lefiNonDefault::layerDiagWidth(i));
   if (def->lefiNonDefault::hasLayerWireExtension(i))
      fprintf(fout, "    WIREEXTENSION %g ;\n",
              def->lefiNonDefault::layerWireExtension(i));
   if (def->lefiNonDefault::hasLayerResistance(i))
      fprintf(fout, "    RESISTANCE RPERSQ %g ;\n",
              def->lefiNonDefault::layerResistance(i));
   if (def->lefiNonDefault::hasLayerCapacitance(i))
      fprintf(fout, "    CAPACITANCE CPERSQDIST %g ;\n",
              def->lefiNonDefault::layerCapacitance(i));
   if (def->lefiNonDefault::hasLayerEdgeCap(i))
      fprintf(fout, "    EDGECAPACITANCE %g ;\n",
              def->lefiNonDefault::layerEdgeCap(i));
   fprintf(fout, "  END %s\n", def->lefiNonDefault::layerName(i));
}

// handle via in nondefaultrule
for (i = 0; i < def->lefiNonDefault::numVias(); i++) {
   via = def->lefiNonDefault::viaRule(i);
   lefVia(via);
}

// handle spacing in nondefaultrule
```

```
for (i = 0; i < def->lefiNonDefault::numSpacingRules(); i++) {
   spacing = def->lefiNonDefault::spacingRule(i);
   lefSpacing(spacing);
}

// handle usevia
for (i = 0; i < def->lefiNonDefault::numUseVia(); i++)
   fprintf(fout, "    USEVIA %s ;\n", def->lefiNonDefault::viaName(i));

// handle useviarule
for (i = 0; i < def->lefiNonDefault::numUseViaRule(); i++)
   fprintf(fout, "    USEVIARULE %s ;\n",
           def->lefiNonDefault::viaRuleName(i));

// handle mincuts
for (i = 0; i < def->lefiNonDefault::numMinCuts(); i++) {
   fprintf(fout, "   MINCUTS %s %d ;\n", def->lefiNonDefault::cutLayerName(i),
           def->lefiNonDefault::numCuts(i));
}

// handle property in nondefaultrule
if (def->lefiNonDefault::numProps() > 0) {
   fprintf(fout, "   PROPERTY ");
   for (i = 0; i < def->lefiNonDefault::numProps(); i++) {
      fprintf(fout, "%s ", def->lefiNonDefault::propName(i));
      if (def->lefiNonDefault::propIsNumber(i))
          fprintf(fout, "%g ", def->lefiNonDefault::propNumber(i));
      if (def->lefiNonDefault::propIsString(i))
          fprintf(fout, "%s ", def->lefiNonDefault::propValue(i));
      switch(def->lefiNonDefault::propType(i)) {
          case 'R': fprintf(fout, "REAL ");
                    break;
          case 'I': fprintf(fout, "INTEGER ");
                    break;
          case 'S': fprintf(fout, "STRING ");
                    break;
          case 'Q': fprintf(fout, "QUOTESTRING ");
                    break;
          case 'N': fprintf(fout, "NUMBER ");
                    break;
      }
```

```
      }
      fprintf(fout, ";\n");
   }
   fprintf(fout, "END %s ;\n", def->lefiNonDefault::name());

   return 0;
}

int obstructionCB(lefrCallbackType_e c, lefiObstruction* obs,
                  lefiUserData ud) {
  lefiGeometries* geometry;

  checkType(c);

  fprintf(fout, "  OBS\n");
  geometry = obs->lefiObstruction::geometries();
  prtGeometry(geometry);
  fprintf(fout, "  END\n");
  return 0;
}

int pinCB(lefrCallbackType_e c, lefiPin* pin, lefiUserData ud) {
  int                  numPorts, i, j;
  lefiGeometries*      geometry;
  lefiPinAntennaModel* aModel;

  checkType(c);

  fprintf(fout, "  PIN %s\n", pin->lefiPin::name());
  if (pin->lefiPin::hasForeign()) {
     for (i = 0; i < pin->lefiPin::numForeigns(); i++) {
        if (pin->lefiPin::hasForeignOrient(i))
           fprintf(fout, "    FOREIGN %s STRUCTURE ( %g %g ) %s ;\n",
                   pin->lefiPin::foreignName(i), pin->lefiPin::foreignX(i),
                   pin->lefiPin::foreignY(i),
                   pin->lefiPin::foreignOrientStr(i));
        else if (pin->lefiPin::hasForeignPoint(i))
           fprintf(fout, "    FOREIGN %s STRUCTURE ( %g %g ) ;\n",
                   pin->lefiPin::foreignName(i), pin->lefiPin::foreignX(i),
                   pin->lefiPin::foreignY(i));
        else
```

```
            fprintf(fout, "    FOREIGN %s ;\n", pin->lefiPin::foreignName(i));
      }
   }
   if (pin->lefiPin::hasLEQ())
      fprintf(fout, "    LEQ %s ;\n", pin->lefiPin::LEQ());
   if (pin->lefiPin::hasDirection())
      fprintf(fout, "    DIRECTION %s ;\n", pin->lefiPin::direction());
   if (pin->lefiPin::hasUse())
      fprintf(fout, "    USE %s ;\n", pin->lefiPin::use());
   if (pin->lefiPin::hasShape())
      fprintf(fout, "    SHAPE %s ;\n", pin->lefiPin::shape());
   if (pin->lefiPin::hasMustjoin())
      fprintf(fout, "    MUSTJOIN %s ;\n", pin->lefiPin::mustjoin());
   if (pin->lefiPin::hasOutMargin())
      fprintf(fout, "    OUTPUTNOISEMARGIN %g %g ;\n",
              pin->lefiPin::outMarginHigh(), pin->lefiPin::outMarginLow());
   if (pin->lefiPin::hasOutResistance())
      fprintf(fout, "    OUTPUTRESISTANCE %g %g ;\n",
              pin->lefiPin::outResistanceHigh(),
              pin->lefiPin::outResistanceLow());
   if (pin->lefiPin::hasInMargin())
      fprintf(fout, "    INPUTNOISEMARGIN %g %g ;\n",
              pin->lefiPin::inMarginHigh(), pin->lefiPin::inMarginLow());
   if (pin->lefiPin::hasPower())
      fprintf(fout, "    POWER %g ;\n", pin->lefiPin::power());
   if (pin->lefiPin::hasLeakage())
      fprintf(fout, "    LEAKAGE %g ;\n", pin->lefiPin::leakage());
   if (pin->lefiPin::hasMaxload())
      fprintf(fout, "    MAXLOAD %g ;\n", pin->lefiPin::maxload());
   if (pin->lefiPin::hasCapacitance())
      fprintf(fout, "    CAPACITANCE %g ;\n", pin->lefiPin::capacitance());
   if (pin->lefiPin::hasResistance())
      fprintf(fout, "    RESISTANCE %g ;\n", pin->lefiPin::resistance());
   if (pin->lefiPin::hasPulldownres())
      fprintf(fout, "    PULLDOWNRES %g ;\n", pin->lefiPin::pulldownres());
   if (pin->lefiPin::hasTieoffr())
      fprintf(fout, "    TIEOFFR %g ;\n", pin->lefiPin::tieoffr());
   if (pin->lefiPin::hasVHI())
      fprintf(fout, "    VHI %g ;\n", pin->lefiPin::VHI());
   if (pin->lefiPin::hasVLO())
      fprintf(fout, "    VLO %g ;\n", pin->lefiPin::VLO());
```

```
if (pin->lefiPin::hasRiseVoltage())
   fprintf(fout, "    RISEVOLTAGETHRESHOLD %g ;\n",
           pin->lefiPin::riseVoltage());
if (pin->lefiPin::hasFallVoltage())
   fprintf(fout, "    FALLVOLTAGETHRESHOLD %g ;\n",
           pin->lefiPin::fallVoltage());
if (pin->lefiPin::hasRiseThresh())
   fprintf(fout, "    RISETHRESH %g ;\n", pin->lefiPin::riseThresh());
if (pin->lefiPin::hasFallThresh())
   fprintf(fout, "    FALLTHRESH %g ;\n", pin->lefiPin::fallThresh());
if (pin->lefiPin::hasRiseSatcur())
   fprintf(fout, "    RISESATCUR %g ;\n", pin->lefiPin::riseSatcur());
if (pin->lefiPin::hasFallSatcur())
   fprintf(fout, "    FALLSATCUR %g ;\n", pin->lefiPin::fallSatcur());
if (pin->lefiPin::hasRiseSlewLimit())
   fprintf(fout, "    RISESLEWLIMIT %g ;\n", pin->lefiPin::riseSlewLimit());
if (pin->lefiPin::hasFallSlewLimit())
   fprintf(fout, "    FALLSLEWLIMIT %g ;\n", pin->lefiPin::fallSlewLimit());
if (pin->lefiPin::hasCurrentSource())
   fprintf(fout, "    CURRENTSOURCE %s ;\n", pin->lefiPin::currentSource());
if (pin->lefiPin::hasTables())
   fprintf(fout, "    IV_TABLES %s %s ;\n", pin->lefiPin::tableHighName(),
           pin->lefiPin::tableLowName());
if (pin->lefiPin::hasTaperRule())
   fprintf(fout, "    TAPERRULE %s ;\n", pin->lefiPin::taperRule());
if (pin->lefiPin::hasNetExpr())
   fprintf(fout, "    NETEXPR \"%s\" ;\n", pin->lefiPin::netExpr());
if (pin->lefiPin::hasSupplySensitivity())
   fprintf(fout, "    SUPPLYSENSITIVITY %s ;\n",
           pin->lefiPin::supplySensitivity());
if (pin->lefiPin::hasGroundSensitivity())
   fprintf(fout, "    GROUNDSENSITIVITY %s ;\n",
           pin->lefiPin::groundSensitivity());
if (pin->lefiPin::hasAntennaSize()) {
   for (i = 0; i < pin->lefiPin::numAntennaSize(); i++) {
      fprintf(fout, "    ANTENNASIZE %g ", pin->lefiPin::antennaSize(i));
      if (pin->lefiPin::antennaSizeLayer(i))
         fprintf(fout, "LAYER %s ", pin->lefiPin::antennaSizeLayer(i));
      fprintf(fout, ";\n");
   }
}
```

```
if (pin->lefiPin::hasAntennaMetalArea()) {
   for (i = 0; i < pin->lefiPin::numAntennaMetalArea(); i++) {
      fprintf(fout, "    ANTENNAMETALAREA %g ",
         pin->lefiPin::antennaMetalArea(i));
      if (pin->lefiPin::antennaMetalAreaLayer(i))
         fprintf(fout, "LAYER %s ", pin->lefiPin::antennaMetalAreaLayer(i));
      fprintf(fout, ";\n");
   }
}
if (pin->lefiPin::hasAntennaMetalLength()) {
   for (i = 0; i < pin->lefiPin::numAntennaMetalLength(); i++) {
      fprintf(fout, "    ANTENNAMETALLENGTH %g ",
         pin->lefiPin::antennaMetalLength(i));
      if (pin->lefiPin::antennaMetalLengthLayer(i))
         fprintf(fout, "LAYER %s ", pin->lefiPin::antennaMetalLengthLayer(i));
      fprintf(fout, ";\n");
   }
}

if (pin->lefiPin::hasAntennaPartialMetalArea()) {
   for (i = 0; i < pin->lefiPin::numAntennaPartialMetalArea(); i++) {
      fprintf(fout, "    ANTENNAPARTIALMETALAREA %g ",
               pin->lefiPin::antennaPartialMetalArea(i));
      if (pin->lefiPin::antennaPartialMetalAreaLayer(i))
         fprintf(fout, "LAYER %s ",
                  pin->lefiPin::antennaPartialMetalAreaLayer(i));
      fprintf(fout, ";\n");
   }
}

if (pin->lefiPin::hasAntennaPartialMetalSideArea()) {
   for (i = 0; i < pin->lefiPin::numAntennaPartialMetalSideArea(); i++) {
      fprintf(fout, "    ANTENNAPARTIALMETALSIDEAREA %g ",
               pin->lefiPin::antennaPartialMetalSideArea(i));
      if (pin->lefiPin::antennaPartialMetalSideAreaLayer(i))
         fprintf(fout, "LAYER %s ",
                  pin->lefiPin::antennaPartialMetalSideAreaLayer(i));
      fprintf(fout, ";\n");
   }
}
```

```
if (pin->lefiPin::hasAntennaPartialCutArea()) {
   for (i = 0; i < pin->lefiPin::numAntennaPartialCutArea(); i++) {
      fprintf(fout, "    ANTENNAPARTIALCUTAREA %g ",
              pin->lefiPin::antennaPartialCutArea(i));
      if (pin->lefiPin::antennaPartialCutAreaLayer(i))
         fprintf(fout, "LAYER %s ",
                 pin->lefiPin::antennaPartialCutAreaLayer(i));
      fprintf(fout, ";\n");
   }
}

if (pin->lefiPin::hasAntennaDiffArea()) {
   for (i = 0; i < pin->lefiPin::numAntennaDiffArea(); i++) {
      fprintf(fout, "    ANTENNADIFFAREA %g ",
              pin->lefiPin::antennaDiffArea(i));
      if (pin->lefiPin::antennaDiffAreaLayer(i))
         fprintf(fout, "LAYER %s ", pin->lefiPin::antennaDiffAreaLayer(i));
      fprintf(fout, ";\n");
   }
}

for (j = 0; j < pin->lefiPin::numAntennaModel(); j++) {
   aModel = pin->lefiPin::antennaModel(j);

   fprintf(fout, "    ANTENNAMODEL %s ;\n",
           aModel->lefiPinAntennaModel::antennaOxide());

   if (aModel->lefiPinAntennaModel::hasAntennaGateArea()) {
      for (i = 0; i < aModel->lefiPinAntennaModel::numAntennaGateArea(); i++)
      {
         fprintf(fout, "    ANTENNAGATEAREA %g ",
                 aModel->lefiPinAntennaModel::antennaGateArea(i));
         if (aModel->lefiPinAntennaModel::antennaGateAreaLayer(i))
            fprintf(fout, "LAYER %s ",
                    aModel->lefiPinAntennaModel::antennaGateAreaLayer(i));
         fprintf(fout, ";\n");
      }
   }

   if (aModel->lefiPinAntennaModel::hasAntennaMaxAreaCar()) {
      for (i = 0; i < aModel->lefiPinAntennaModel::numAntennaMaxAreaCar();
```

```
         i++) {
         fprintf(fout, "    ANTENNAMAXAREACAR %g ",
                 aModel->lefiPinAntennaModel::antennaMaxAreaCar(i));
         if (aModel->lefiPinAntennaModel::antennaMaxAreaCarLayer(i))
            fprintf(fout, "LAYER %s ",
                 aModel->lefiPinAntennaModel::antennaMaxAreaCarLayer(i));
         fprintf(fout, ";\n");
      }
   }

   if (aModel->lefiPinAntennaModel::hasAntennaMaxSideAreaCar()) {
      for (i = 0; i < aModel->lefiPinAntennaModel::numAntennaMaxSideAreaCar();
         i++) {
         fprintf(fout, "    ANTENNAMAXSIDEAREACAR %g ",
                 aModel->lefiPinAntennaModel::antennaMaxSideAreaCar(i));
         if (aModel->lefiPinAntennaModel::antennaMaxSideAreaCarLayer(i))
            fprintf(fout, "LAYER %s ",
                 aModel->lefiPinAntennaModel::antennaMaxSideAreaCarLayer(i));
         fprintf(fout, ";\n");
      }
   }

   if (aModel->lefiPinAntennaModel::hasAntennaMaxCutCar()) {
      for (i = 0; i < aModel->lefiPinAntennaModel::numAntennaMaxCutCar(); i++)
      {
         fprintf(fout, "    ANTENNAMAXCUTCAR %g ",
                 aModel->lefiPinAntennaModel::antennaMaxCutCar(i));
         if (aModel->lefiPinAntennaModel::antennaMaxCutCarLayer(i))
            fprintf(fout, "LAYER %s ",
                 aModel->lefiPinAntennaModel::antennaMaxCutCarLayer(i));
         fprintf(fout, ";\n");
      }
   }
}

if (pin->lefiPin::numProperties() > 0) {
   fprintf(fout, "    PROPERTY ");
   for (i = 0; i < pin->lefiPin::numProperties(); i++) {
      // value can either be a string or number
      if (pin->lefiPin::propValue(i)) {
         fprintf(fout, "%s %s ", pin->lefiPin::propName(i),
```

```
                 pin->lefiPin::propValue(i));
        }
        else
           fprintf(fout, "%s %g ", pin->lefiPin::propName(i),
                   pin->lefiPin::propNum(i));
        switch (pin->lefiPin::propType(i)) {
           case 'R': fprintf(fout, "REAL ");
                     break;
           case 'I': fprintf(fout, "INTEGER ");
                     break;
           case 'S': fprintf(fout, "STRING ");
                     break;
           case 'Q': fprintf(fout, "QUOTESTRING ");
                     break;
           case 'N': fprintf(fout, "NUMBER ");
                     break;
        }
     }
     fprintf(fout, ";\n");
  }

  numPorts = pin->lefiPin::numPorts();
  for (i = 0; i < numPorts; i++) {
     fprintf(fout,"    PORT\n");
     geometry = pin->lefiPin::port(i);
     prtGeometry(geometry);
     fprintf(fout, "    END\n");
  }
  fprintf(fout, "  END %s\n", pin->lefiPin::name());
  return 0;
}

int densityCB(lefrCallbackType_e c, lefiDensity* density,
                  lefiUserData ud) {

  struct lefiGeomRect rect;

  checkType(c);

  fprintf(fout, "  DENSITY\n");
  for (int i = 0; i < density->lefiDensity::numLayer(); i++) {
```

```
    fprintf(fout, "    LAYER %s ;\n", density->lefiDensity::layerName(i));
    for (int j = 0; j < density->lefiDensity::numRects(i); j++) {
      rect = density->lefiDensity::getRect(i,j);
      fprintf(fout, "      RECT %g %g %g %g ", rect.xl, rect.yl, rect.xh,
              rect.yh);
      fprintf(fout, "%g ;\n", density->lefiDensity::densityValue(i,j));
    }
  }
  fprintf(fout, "  END\n");
  return 0;
}

int propDefBeginCB(lefrCallbackType_e c, void* ptr, lefiUserData ud) {

  checkType(c);

  fprintf(fout, "PROPERTYDEFINITIONS\n");
  return 0;
}

int propDefCB(lefrCallbackType_e c, lefiProp* prop, lefiUserData ud) {

  checkType(c);

  fprintf(fout, " %s %s", prop->lefiProp::propType(),
          prop->lefiProp::propName());
  switch(prop->lefiProp::dataType()) {
     case 'I':
          fprintf(fout, " INTEGER");
          break;
     case 'R':
          fprintf(fout, " REAL");
          break;
     case 'S':
          fprintf(fout, " STRING");
          break;
  }
  if (prop->lefiProp::hasNumber())
     fprintf(fout, " %g", prop->lefiProp::number());
  if (prop->lefiProp::hasRange())
     fprintf(fout, " RANGE %g %g", prop->lefiProp::left(),
```

```
            prop->lefiProp::right());
  if (prop->lefiProp::hasString())
     fprintf(fout, " %s", prop->lefiProp::string());
  fprintf(fout, "\n");
  return 0;
}

int propDefEndCB(lefrCallbackType_e c, void* ptr, lefiUserData ud) {

  checkType(c);

  fprintf(fout, "END PROPERTYDEFINITIONS\n");
  return 0;
}

int siteCB(lefrCallbackType_e c, lefiSite* site, lefiUserData ud) {
  int hasPrtSym = 0;
  int i;

  checkType(c);

  fprintf(fout, "SITE %s\n", site->lefiSite::name());
  if (site->lefiSite::hasClass())
     fprintf(fout, "  CLASS %s ;\n", site->lefiSite::siteClass());
  if (site->lefiSite::hasXSymmetry()) {
     fprintf(fout, "  SYMMETRY X ");
     hasPrtSym = 1;
  }
  if (site->lefiSite::hasYSymmetry()) {
     if (hasPrtSym)
        fprintf(fout, "Y ");
     else {
        fprintf(fout, "  SYMMETRY Y ");
        hasPrtSym = 1;
     }
  }
  if (site->lefiSite::has90Symmetry()) {
     if (hasPrtSym)
        fprintf(fout, "R90 ");
     else {
        fprintf(fout, "  SYMMETRY R90 ");
```

```
        hasPrtSym = 1;
      }
    }
    if (hasPrtSym)
      fprintf(fout, ";\n");
    if (site->lefiSite::hasSize())
      fprintf(fout, "  SIZE %g BY %g ;\n", site->lefiSite::sizeX(),
              site->lefiSite::sizeY());

    if (site->hasRowPattern()) {
      fprintf(fout, "  ROWPATTERN ");
      for (i = 0; i < site->lefiSite::numSites(); i++)
        fprintf(fout, "  %s %s ", site->lefiSite::siteName(i),
                site->lefiSite::siteOrientStr(i));
      fprintf(fout, ";\n");
    }

    fprintf(fout, "END %s\n", site->lefiSite::name());
    return 0;
}

int spacingBeginCB(lefrCallbackType_e c, void* ptr, lefiUserData ud){
  checkType(c);

  fprintf(fout, "SPACING\n");
  return 0;
}

int spacingCB(lefrCallbackType_e c, lefiSpacing* spacing, lefiUserData ud) {
  checkType(c);

  lefSpacing(spacing);
  return 0;
}

int spacingEndCB(lefrCallbackType_e c, void* ptr, lefiUserData ud){
  checkType(c);

  fprintf(fout, "END SPACING\n");
  return 0;
}
```

```
int timingCB(lefrCallbackType_e c, lefiTiming* timing, lefiUserData ud) {
  int i;
  checkType(c);

  fprintf(fout, "TIMING\n");
  for (i = 0; i < timing->numFromPins(); i++)
     fprintf(fout, " FROMPIN %s ;\n", timing->fromPin(i));
  for (i = 0; i < timing->numToPins(); i++)
     fprintf(fout, " TOPIN %s ;\n", timing->toPin(i));
     fprintf(fout, " RISE SLEW1 %g %g %g %g ;\n", timing->riseSlewOne(),
             timing->riseSlewTwo(), timing->riseSlewThree(),
             timing->riseSlewFour());
  if (timing->hasRiseSlew2())
     fprintf(fout, " RISE SLEW2 %g %g %g ;\n", timing->riseSlewFive(),
             timing->riseSlewSix(), timing->riseSlewSeven());
  if (timing->hasFallSlew())
     fprintf(fout, " FALL SLEW1 %g %g %g %g ;\n", timing->fallSlewOne(),
             timing->fallSlewTwo(), timing->fallSlewThree(),
             timing->fallSlewFour());
  if (timing->hasFallSlew2())
     fprintf(fout, " FALL SLEW2 %g %g %g ;\n", timing->fallSlewFive(),
             timing->fallSlewSix(), timing->riseSlewSeven());
  if (timing->hasRiseIntrinsic()) {
     fprintf(fout, "TIMING RISE INTRINSIC %g %g ;\n",
             timing->riseIntrinsicOne(), timing->riseIntrinsicTwo());
     fprintf(fout, "TIMING RISE VARIABLE %g %g ;\n",
             timing->riseIntrinsicThree(), timing->riseIntrinsicFour());
  }
  if (timing->hasFallIntrinsic()) {
     fprintf(fout, "TIMING FALL INTRINSIC %g %g ;\n",
             timing->fallIntrinsicOne(), timing->fallIntrinsicTwo());
     fprintf(fout, "TIMING RISE VARIABLE %g %g ;\n",
             timing->fallIntrinsicThree(), timing->fallIntrinsicFour());
  }
  if (timing->hasRiseRS())
     fprintf(fout, "TIMING RISERS %g %g ;\n",
             timing->riseRSOne(), timing->riseRSTwo());
     if (timing->hasRiseCS())
     fprintf(fout, "TIMING RISECS %g %g ;\n",
             timing->riseCSOne(), timing->riseCSTwo());
```

```
  if (timing->hasFallRS())
     fprintf(fout, "TIMING FALLRS %g %g ;\n",
             timing->fallRSOne(), timing->fallRSTwo());
  if (timing->hasFallCS())
     fprintf(fout, "TIMING FALLCS %g %g ;\n",
             timing->fallCSOne(), timing->fallCSTwo());
  if (timing->hasUnateness())
     fprintf(fout, "TIMING UNATENESS %s ;\n", timing->unateness());
  if (timing->hasRiseAtt1())
     fprintf(fout, "TIMING RISESATT1 %g %g ;\n", timing->riseAtt1One(),
             timing->riseAtt1Two());
  if (timing->hasFallAtt1())
     fprintf(fout, "TIMING FALLSATT1 %g %g ;\n", timing->fallAtt1One(),
             timing->fallAtt1Two());
  if (timing->hasRiseTo())
     fprintf(fout, "TIMING RISET0 %g %g ;\n", timing->riseToOne(),
             timing->riseToTwo());
  if (timing->hasFallTo())
     fprintf(fout, "TIMING FALLT0 %g %g ;\n", timing->fallToOne(),
             timing->fallToTwo());
  if (timing->hasSDFonePinTrigger())
     fprintf(fout, " %s TABLEDIMENSION %g %g %g ;\n",
             timing->SDFonePinTriggerType(), timing->SDFtriggerOne(),
             timing->SDFtriggerTwo(), timing->SDFtriggerThree());
  if (timing->hasSDFtwoPinTrigger())
     fprintf(fout, " %s %s %s TABLEDIMENSION %g %g %g ;\n",
             timing->SDFtwoPinTriggerType(), timing->SDFfromTrigger(),
             timing->SDFtoTrigger(), timing->SDFtriggerOne(),
             timing->SDFtriggerTwo(), timing->SDFtriggerThree());
  fprintf(fout, "END TIMING\n");
  return 0;
}

int unitsCB(lefrCallbackType_e c, lefiUnits* unit, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "UNITS\n");
  if (unit->lefiUnits::hasDatabase())
     fprintf(fout, "  DATABASE %s %g ;\n", unit->lefiUnits::databaseName(),
             unit->lefiUnits::databaseNumber());
  if (unit->lefiUnits::hasCapacitance())
```

```
     fprintf(fout, "  CAPACITANCE PICOFARADS %g ;\n",
             unit->lefiUnits::capacitance());
  if (unit->lefiUnits::hasResistance())
     fprintf(fout, "  RESISTANCE OHMS %g ;\n", unit->lefiUnits::resistance());
  if (unit->lefiUnits::hasPower())
     fprintf(fout, "  POWER MILLIWATTS %g ;\n", unit->lefiUnits::power());
  if (unit->lefiUnits::hasCurrent())
     fprintf(fout, "  CURRENT MILLIAMPS %g ;\n", unit->lefiUnits::current());
  if (unit->lefiUnits::hasVoltage())
     fprintf(fout, "  VOLTAGE VOLTS %g ;\n", unit->lefiUnits::voltage());
  if (unit->lefiUnits::hasFrequency())
     fprintf(fout, "  FREQUENCY MEGAHERTZ %g ;\n",
             unit->lefiUnits::frequency());
  fprintf(fout, "END UNITS\n");
  return 0;
}

int useMinSpacingCB(lefrCallbackType_e c, lefiUseMinSpacing* spacing,
                    lefiUserData ud) {
  checkType(c);

  fprintf(fout, "USEMINSPACING %s ", spacing->lefiUseMinSpacing::name());
  if (spacing->lefiUseMinSpacing::value())
      fprintf(fout, "ON ;\n");
  else
      fprintf(fout, "OFF ;\n");
  return 0;
}

int versionCB(lefrCallbackType_e c, double num, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "VERSION %g ;\n", num);
  return 0;
}

int versionStrCB(lefrCallbackType_e c, const char* versionName, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "VERSION %s ;\n", versionName);
  return 0;
```

```
}

int viaCB(lefrCallbackType_e c, lefiVia* via, lefiUserData ud) {
  checkType(c);

  lefVia(via);
  return 0;
}

int viaRuleCB(lefrCallbackType_e c, lefiViaRule* viaRule, lefiUserData ud) {
  int                numLayers, numVias, i;
  lefiViaRuleLayer* vLayer;

  checkType(c);

  fprintf(fout, "VIARULE %s", viaRule->lefiViaRule::name());
  if (viaRule->lefiViaRule::hasGenerate())
     fprintf(fout, " GENERATE");
  if (viaRule->lefiViaRule::hasDefault())
     fprintf(fout, " DEFAULT");
  fprintf(fout, "\n");

  numLayers = viaRule->lefiViaRule::numLayers();
  for (i = 0; i < numLayers; i++) {
     vLayer = viaRule->lefiViaRule::layer(i);
     lefViaRuleLayer(vLayer);
  }

  if (numLayers == 2 && !(viaRule->lefiViaRule::hasGenerate())) {
     numVias = viaRule->lefiViaRule::numVias();
     if (numVias == 0)
        fprintf(fout, "Should have via names in VIARULE.\n");
     else {
        for (i = 0; i < numVias; i++)
           fprintf(fout, "  VIA %s ;\n", viaRule->lefiViaRule::viaName(i));
     }
  }
  if (viaRule->lefiViaRule::numProps() > 0) {
     fprintf(fout, "  PROPERTY ");
     for (i = 0; i < viaRule->lefiViaRule::numProps(); i++) {
        fprintf(fout, "%s ", viaRule->lefiViaRule::propName(i));
```

```
        if (viaRule->lefiViaRule::propValue(i))
           fprintf(fout, "%s ", viaRule->lefiViaRule::propValue(i));
        switch (viaRule->lefiViaRule::propType(i)) {
           case 'R': fprintf(fout, "REAL ");
                     break;
           case 'I': fprintf(fout, "INTEGER ");
                     break;
           case 'S': fprintf(fout, "STRING ");
                     break;
           case 'Q': fprintf(fout, "QUOTESTRING ");
                     break;
           case 'N': fprintf(fout, "NUMBER ");
                     break;
        }
     }
     fprintf(fout, ";\n");
  }
  fprintf(fout, "END %s\n", viaRule->lefiViaRule::name());
  return 0;
}

int extensionCB(lefrCallbackType_e c, const char* extsn, lefiUserData ud) {
  checkType(c);
  fprintf(fout, "BEGINEXT %s ;\n", extsn);
  return 0;
}

int doneCB(lefrCallbackType_e c, void* ptr, lefiUserData ud) {
  checkType(c);

  fprintf(fout, "END LIBRARY\n");
  return 0;
}

void errorCB(const char* msg) {
  printf ("%s : %s\n", lefrGetUserData(), msg);
}

void warningCB(const char* msg) {
  printf ("%s : %s\n", lefrGetUserData(), msg);
}
```

```
void* mallocCB(int size) {
  return malloc(size);
}

void* reallocCB(void* name, int size) {
  return realloc(name, size);
}

void freeCB(void* name) {
  free(name);
  return;
}

void lineNumberCB(int lineNo) {
  fprintf(fout, "Parsed %d number of lines!!\n", lineNo);
  return;
}

int
main(int argc, char** argv) {
  char* inFile[100];
  char* outFile;
  FILE* f;
  int res;
  int noCalls = 0;
  int num;
  int status;
  int retStr = 0;
  int numInFile = 0;
  int fileCt = 0;
  int relax = 0;
  char* version;
  int setVer = 0;
  char* userData;
  int msgCb = 0;

  userData = strdup ("(lefrw-5100)");
  strcpy(defaultName,"lef.in");
  strcpy(defaultOut,"list");
  inFile[0] = defaultName;
```

```
outFile = defaultOut;
fout = stdout;

argc--;
argv++;
while (argc--) {

  if (strcmp(*argv, "-d") == 0) {
    argv++;
    argc--;
    sscanf(*argv, "%d", &num);
    lefiSetDebug(num, 1);

  } else if (strcmp(*argv, "-nc") == 0) {
    noCalls = 1;

  } else if (strcmp(*argv, "-p") == 0) {
    printing = 1;

  } else if (strcmp(*argv, "-m") == 0) { // use the user error/warning CB
    msgCb = 1;

  } else if (strcmp(*argv, "-o") == 0) {
    argv++;
    argc--;
    outFile = *argv;
    if ((fout = fopen(outFile, "w")) == 0) {
  fprintf(stderr, "ERROR: could not open output file\n");
  return 2;
    }

  } else if (strcmp(*argv, "-verStr") == 0) {
      /* New to set the version callback routine to return a string    */
      /* instead of double.                                             */
      retStr = 1;

  } else if (strcmp(*argv, "-relax") == 0) {
      relax = 1;

  } else if (strcmp(*argv, "-65nm") == 0) {
      parse65nm = 1;
```

```
    } else if (strcmp(*argv, "-ver") == 0) {
      argv++;
      argc--;
      setVer = 1;
      version = *argv;

    } else if (argv[0][0] != '-') {
      if (numInFile >= 100) {
        fprintf(stderr, "ERROR: too many input files, max = 3.\n");
        return 2;
      }
      inFile[numInFile++] = *argv;

    } else {
      fprintf(stderr, "ERROR: Illegal command line option: '%s'\n", *argv);
      return 2;
    }

    argv++;
  }

  if (noCalls == 0) {
     lefrSetAntennaInputCbk(antennaCB);
     lefrSetAntennaInoutCbk(antennaCB);
     lefrSetAntennaOutputCbk(antennaCB);
     lefrSetArrayBeginCbk(arrayBeginCB);
     lefrSetArrayCbk(arrayCB);
     lefrSetArrayEndCbk(arrayEndCB);
     lefrSetBusBitCharsCbk(busBitCharsCB);
     lefrSetCaseSensitiveCbk(caseSensCB);
     lefrSetClearanceMeasureCbk(clearanceCB);
     lefrSetDensityCbk(densityCB);
     lefrSetDividerCharCbk(dividerCB);
     lefrSetNoWireExtensionCbk(noWireExtCB);
     lefrSetEdgeRateThreshold1Cbk(edge1CB);
     lefrSetEdgeRateThreshold2Cbk(edge2CB);
     lefrSetEdgeRateScaleFactorCbk(edgeScaleCB);
     lefrSetExtensionCbk(extensionCB);
     lefrSetDielectricCbk(dielectricCB);
     lefrSetIRDropBeginCbk(irdropBeginCB);
```

```
lefrSetIRDropCbk(irdropCB);
lefrSetIRDropEndCbk(irdropEndCB);
lefrSetLayerCbk(layerCB);
lefrSetLibraryEndCbk(doneCB);
lefrSetMacroBeginCbk(macroBeginCB);
lefrSetMacroCbk(macroCB);
lefrSetMacroClassTypeCbk(macroClassTypeCB);
lefrSetMacroEndCbk(macroEndCB);
lefrSetManufacturingCbk(manufacturingCB);
lefrSetMaxStackViaCbk(maxStackViaCB);
lefrSetMinFeatureCbk(minFeatureCB);
lefrSetNonDefaultCbk(nonDefaultCB);
lefrSetObstructionCbk(obstructionCB);
lefrSetPinCbk(pinCB);
lefrSetPropBeginCbk(propDefBeginCB);
lefrSetPropCbk(propDefCB);
lefrSetPropEndCbk(propDefEndCB);
lefrSetSiteCbk(siteCB);
lefrSetSpacingBeginCbk(spacingBeginCB);
lefrSetSpacingCbk(spacingCB);
lefrSetSpacingEndCbk(spacingEndCB);
lefrSetTimingCbk(timingCB);
lefrSetUnitsCbk(unitsCB);
lefrSetUseMinSpacingCbk(useMinSpacingCB);
lefrSetUserData((void*)3);
if (!retStr)
  lefrSetVersionCbk(versionCB);
else
  lefrSetVersionStrCbk(versionStrCB);
lefrSetViaCbk(viaCB);
lefrSetViaRuleCbk(viaRuleCB);
lefrSetInputAntennaCbk(antennaCB);
lefrSetOutputAntennaCbk(antennaCB);
lefrSetInoutAntennaCbk(antennaCB);

if (msgCb) {
  lefrSetLogFunction(errorCB);
  lefrSetWarningLogFunction(warningCB);
}

lefrSetMallocFunction(mallocCB);
```

```
lefrSetReallocFunction(reallocCB);
lefrSetFreeFunction(freeCB);

lefrSetLineNumberFunction(lineNumberCB);
lefrSetDeltaNumberLines(50);

lefrSetRegisterUnusedCallbacks();

if (relax)
  lefrSetRelaxMode();

if (setVer)
  (void)lefrSetVersionValue(version);

lefrSetAntennaInoutWarnings(30);
lefrSetAntennaInputWarnings(30);
lefrSetAntennaOutputWarnings(30);
lefrSetArrayWarnings(30);
lefrSetCaseSensitiveWarnings(30);
lefrSetCorrectionTableWarnings(30);
lefrSetDielectricWarnings(30);
lefrSetEdgeRateThreshold1Warnings(30);
lefrSetEdgeRateThreshold2Warnings(30);
lefrSetEdgeRateScaleFactorWarnings(30);
lefrSetInoutAntennaWarnings(30);
lefrSetInputAntennaWarnings(30);
lefrSetIRDropWarnings(30);
lefrSetLayerWarnings(30);
lefrSetMacroWarnings(30);
lefrSetMaxStackViaWarnings(30);
lefrSetMinFeatureWarnings(30);
lefrSetNoiseMarginWarnings(30);
lefrSetNoiseTableWarnings(30);
lefrSetNonDefaultWarnings(30);
lefrSetNoWireExtensionWarnings(30);
lefrSetOutputAntennaWarnings(30);
lefrSetPinWarnings(30);
lefrSetSiteWarnings(30);
lefrSetSpacingWarnings(30);
lefrSetTimingWarnings(30);
lefrSetUnitsWarnings(30);
```

```
    lefrSetUseMinSpacingWarnings(30);
    lefrSetViaRuleWarnings(30);
    lefrSetViaWarnings(30);
  }

  (void) lefrSetShiftCase();  // will shift name to uppercase if caseinsensitive
                              // is set to off or not set

  lefrInit();

  for (fileCt = 0; fileCt < numInFile; fileCt++) {
    lefrReset();

    if ((f = fopen(inFile[fileCt],"r")) == 0) {
      fprintf(stderr,"Couldn't open input file '%s'\n", inFile[fileCt]);
      return(2);
    }

    (void)lefrEnableReadEncrypted();

    status = lefwInit(fout); // initialize the lef writer, need to be called 1st
    if (status != LEFW_OK)
       return 1;

    res = lefrRead(f, inFile[fileCt], (void*)userData);

    if (res)
       fprintf(stderr, "Reader returns bad status.\n", inFile[fileCt]);

    (void)lefrPrintUnusedCallbacks(fout);
    (void)lefrReleaseNResetMemory();

  }
  fclose(fout);

  return 0;
}
```

LEF Writer Program

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#ifndef WIN32
#   include <unistd.h>
#endif /* not WIN32 */
#include "lefwWriter.hpp"

char defaultOut[128];

// Global variables
FILE* fout;

#define CHECK_STATUS(status) \
  if (status) {              \
     lefwPrintError(status); \
     return(status);         \
  }

int main(int argc, char** argv) {
  char* outfile;
  int   status;    // return code, if none 0 means error
  int   lineNum = 0;

  // assign the default
  strcpy(defaultOut, "lef.in");
  outfile = defaultOut;
  fout = stdout;

  double *xpath;
  double *ypath;
  double *xl;
  double *yl;
  double *wthn, *spng;

  argc--;
  argv++;
  while (argc--) {
     if (strcmp(*argv, "-o") == 0) {   // output filename
```

```
        argv++;
        argc--;
        outfile = *argv;
        if ((fout = fopen(outfile, "w")) == 0) {
           fprintf(stderr, "ERROR: could not open output file\n");
           return 2;
        }
     } else if (strncmp(*argv,  "-h", 2) == 0) {  // compare with -h[elp]
        fprintf(stderr, "Usage: lefwrite [-o <filename>] [-help]\n");
        return 1;
     } else {
        fprintf(stderr, "ERROR: Illegal command line option: '%s'\n", *argv);
        return 2;
     }
     argv++;
  }

  // initalize
  status = lefwInit(fout);
  CHECK_STATUS(status);
  status = lefwVersion(5, 7);
  CHECK_STATUS(status);
  status = lefwBusBitChars("<>");
  CHECK_STATUS(status);
  status = lefwDividerChar(":");
  CHECK_STATUS(status);
  status = lefwManufacturingGrid(3.5);
  CHECK_STATUS(status);
  status = lefwUseMinSpacing("OBS", "OFF");
  CHECK_STATUS(status);
  status = lefwClearanceMeasure("EUCLIDEAN");
  CHECK_STATUS(status);
  status = lefwNewLine();
  CHECK_STATUS(status);

  // 5.4 ANTENNA
  status = lefwAntennaInputGateArea(45);
  CHECK_STATUS(status);
  status = lefwAntennaInOutDiffArea(65);
  CHECK_STATUS(status);
  status = lefwAntennaOutputDiffArea(55);
```

```
CHECK_STATUS(status);
status = lefwNewLine();
CHECK_STATUS(status);

// UNITS
status = lefwStartUnits();
CHECK_STATUS(status);
status = lefwUnits(100, 10, 10000, 10000, 10000, 1000, 20000);
CHECK_STATUS(status);
status = lefwUnitsFrequency(10);
CHECK_STATUS(status);
status = lefwEndUnits();
CHECK_STATUS(status);

// PROPERTYDEFINITIONS
status = lefwStartPropDef();
CHECK_STATUS(status);
status = lefwStringPropDef("LIBRARY", "NAME", 0, 0, "Cadence96");
CHECK_STATUS(status);
status = lefwIntPropDef("LIBRARY", "intNum", 0, 0, 20);
CHECK_STATUS(status);
status = lefwRealPropDef("LIBRARY", "realNum", 0, 0, 21.22);
CHECK_STATUS(status);
status = lefwStringPropDef("PIN", "TYPE", 0, 0, 0);
CHECK_STATUS(status);
status = lefwIntPropDef("PIN", "intProp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwRealPropDef("PIN", "realProp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwStringPropDef("MACRO", "stringProp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwIntPropDef("MACRO", "integerProp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwRealPropDef("MACRO", "WEIGHT", 1.0, 100.0, 0);
CHECK_STATUS(status);
status = lefwStringPropDef("VIA", "stringProperty", 0, 0, 0);
CHECK_STATUS(status);
status = lefwRealPropDef("VIA", "realProp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwIntPropDef("VIA", "COUNT", 1, 100, 0);
CHECK_STATUS(status);
```

```
status = lefwStringPropDef("LAYER", "lsp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwIntPropDef("LAYER", "lip", 0, 0, 0);
CHECK_STATUS(status);
status = lefwRealPropDef("LAYER", "lrp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwStringPropDef("VIARULE", "vrsp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwRealPropDef("VIARULE", "vrip", 0, 0, 0);
CHECK_STATUS(status);
status = lefwIntPropDef("VIARULE", "vrrp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwStringPropDef("NONDEFAULTRULE", "ndrsp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwIntPropDef("NONDEFAULTRULE", "ndrip", 0, 0, 0);
CHECK_STATUS(status);
status = lefwRealPropDef("NONDEFAULTRULE", "ndrrp", 0, 0, 0);
CHECK_STATUS(status);
status = lefwEndPropDef();
CHECK_STATUS(status);

// LAYERS
double *current;
double *diffs;
double *ratios;
double *area;
double *width;

current = (double*)malloc(sizeof(double)*15);
diffs = (double*)malloc(sizeof(double)*15);
ratios = (double*)malloc(sizeof(double)*15);

status = lefwStartLayer("POLYS", "MASTERSLICE");
CHECK_STATUS(status);
status = lefwStringProperty("lsp", "top");
CHECK_STATUS(status);
status = lefwIntProperty("lip", 1);
CHECK_STATUS(status);
status = lefwRealProperty("lrp", 2.3);
CHECK_STATUS(status);
status = lefwEndLayer("POLYS");
```

```
CHECK_STATUS(status);

status = lefwStartLayer("CUT01", "CUT");
CHECK_STATUS(status);
status = lefwLayerDCCurrentDensity("AVERAGE", 0);
CHECK_STATUS(status);
current[0] = 2.0;
current[1] = 5.0;
current[2] = 10.0;
status = lefwLayerDCCutarea(3, current);
CHECK_STATUS(status);
current[0] = 0.6E-6;
current[1] = 0.5E-6;
current[2] = 0.4E-6;
status = lefwLayerDCTableEntries(3, current);
CHECK_STATUS(status);
status = lefwEndLayer("CUT01");
CHECK_STATUS(status);

status = lefwStartLayerRouting("RX");
CHECK_STATUS(status);
status = lefwLayerRouting("HORIZONTAL", 1);
CHECK_STATUS(status);
status = lefwLayerRoutingPitch(1.8);
CHECK_STATUS(status);
status = lefwLayerRoutingDiagPitch(1.5);
CHECK_STATUS(status);
status = lefwLayerRoutingDiagWidth(1.0);
CHECK_STATUS(status);
status = lefwLayerRoutingDiagSpacing(0.05);
CHECK_STATUS(status);
status = lefwLayerRoutingDiagMinEdgeLength(0.07);
CHECK_STATUS(status);
status = lefwLayerRoutingArea(34.1);
CHECK_STATUS(status);
xl = (double*)malloc(sizeof(double)*2);
yl = (double*)malloc(sizeof(double)*2);
xl[0] = 0.14;
yl[0] = 0.30;
xl[1] = 0.08;
yl[1] = 0.33;
```

```
status = lefwLayerRoutingMinsize(2, xl, yl);
CHECK_STATUS(status);
free((char*)xl);
free((char*)yl);
status = lefwLayerRoutingWireExtension(0.75);
CHECK_STATUS(status);
status = lefwLayerRoutingOffset(0.9);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.6);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingRange(0.1, 9);
CHECK_STATUS(status);
status = lefwLayerRoutingResistance("0.103");
CHECK_STATUS(status);
status = lefwLayerRoutingCapacitance("0.000156");
CHECK_STATUS(status);
status = lefwLayerRoutingHeight(9);
CHECK_STATUS(status);
status = lefwLayerRoutingThickness(1);
CHECK_STATUS(status);
status = lefwLayerRoutingShrinkage(0.1);
CHECK_STATUS(status);
status = lefwLayerRoutingEdgeCap(0.00005);
CHECK_STATUS(status);
status = lefwLayerRoutingCapMultiplier(1);
CHECK_STATUS(status);
status = lefwLayerRoutingMinwidth(0.15);
CHECK_STATUS(status);
status = lefwLayerRoutingAntennaArea(1);
CHECK_STATUS(status);
status = lefwLayerAntennaCumAreaRatio(6.7);              // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaCumRoutingPlusCut();            // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaAreaMinusDiff(100.0);           // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaGatePlusDiff(2.0);              // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaCumDiffAreaRatio(1000);         // 5.7
CHECK_STATUS(status);
xl = (double*)malloc(sizeof(double)*5);
```

```
yl = (double*)malloc(sizeof(double)*5);
xl[0] = 0.0;
yl[0] = 1.0;
xl[1] = 0.09999;
yl[1] = 1.0;
xl[2] = 0.1;
yl[2] = 0.2;
xl[3] = 1.0;
yl[3] = 0.1;
xl[4] = 100;
yl[4] = 0.1;
status = lefwLayerAntennaAreaDiffReducePwl(5, xl, yl);   // 5.7
CHECK_STATUS(status);
free((char*)xl);
free((char*)yl);
status = lefwLayerAntennaCumDiffAreaRatio(1000);         // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingAntennaLength(1);
CHECK_STATUS(status);
status = lefwLayerACCurrentDensity("PEAK", 0);
CHECK_STATUS(status);
current[0] = 1E6;
current[1] = 100E6;
current[2] = 400E6;
status = lefwLayerACFrequency(3, current);
CHECK_STATUS(status);
current[0] = 0.4;
current[1] = 0.8;
current[2] = 10.0;
current[3] = 50.0;
status = lefwLayerACCutarea(4, current);
CHECK_STATUS(status);
current[0] = 0.4;
current[1] = 0.8;
current[2] = 10.0;
current[3] = 50.0;
current[4] = 100.0;
status = lefwLayerACWidth(5, current);
CHECK_STATUS(status);
current[0] = 2.0E-6;
current[1] = 1.9E-6;
```

```
current[2] = 1.8E-6;
current[3] = 1.7E-6;
current[4] = 1.5E-6;
current[5] = 1.4E-6;
current[6] = 1.3E-6;
current[7] = 1.2E-6;
current[8] = 1.1E-6;
current[9] = 1.0E-6;
current[10] = 0.9E-6;
current[11] = 0.8E-6;
current[12] = 0.7E-6;
current[13] = 0.6E-6;
current[14] = 0.4E-6;
status = lefwLayerACTableEntries(15, current);
CHECK_STATUS(status);
status = lefwLayerACCurrentDensity("AVERAGE", 0);
CHECK_STATUS(status);
current[0] = 1E6;
current[1] = 100E6;
current[2] = 400E6;
status = lefwLayerACFrequency(3, current);
CHECK_STATUS(status);
current[0] = 0.6E-6;
current[1] = 0.5E-6;
current[2] = 0.4E-6;
status = lefwLayerACTableEntries(3, current);
CHECK_STATUS(status);
status = lefwLayerACCurrentDensity("RMS", 0);
CHECK_STATUS(status);
current[0] = 1E6;
current[1] = 400E6;
current[2] = 800E6;
status = lefwLayerACFrequency(3, current);
CHECK_STATUS(status);
current[0] = 0.4;
current[1] = 0.8;
current[2] = 10.0;
current[3] = 50.0;
current[4] = 100.0;
status = lefwLayerACWidth(5, current);
CHECK_STATUS(status);
```

```
current[0] = 2.0E-6;
current[1] = 1.9E-6;
current[2] = 1.8E-6;
current[3] = 1.7E-6;
current[4] = 1.5E-6;
current[5] = 1.4E-6;
current[6] = 1.3E-6;
current[7] = 1.2E-6;
current[8] = 1.1E-6;
current[9] = 1.0E-6;
current[10] = 0.9E-6;
current[11] = 0.8E-6;
current[12] = 0.7E-6;
current[13] = 0.6E-6;
current[14] = 0.4E-6;
status = lefwLayerACTableEntries(15, current);
CHECK_STATUS(status);
status = lefwEndLayerRouting("RX");
CHECK_STATUS(status);

status = lefwStartLayer("CUT12", "CUT");
CHECK_STATUS(status);
status = lefwLayerCutSpacing(0.7);
CHECK_STATUS(status);
status = lefwLayerCutSpacingLayer("RX", 0);
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();
CHECK_STATUS(status);
status = lefwLayerResistancePerCut(8.0);
CHECK_STATUS(status);
status = lefwLayerCutSpacing(0.22);                // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingAdjacent(3, 0.25, 0);  // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                 // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacing(1.5);                 // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingParallel();            // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                 // 5.7
```

```
CHECK_STATUS(status);
status = lefwLayerCutSpacing(1.2);                  // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingAdjacent(2, 1.5, 0);    // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                  // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaModel("OXIDE1");
CHECK_STATUS(status);
status = lefwLayerAntennaAreaRatio(5.6);
CHECK_STATUS(status);
status = lefwLayerAntennaDiffAreaRatio(6.5);
CHECK_STATUS(status);
status = lefwLayerAntennaAreaFactor(5.4, 0);
CHECK_STATUS(status);
status = lefwLayerAntennaCumDiffAreaRatio(4.5);
CHECK_STATUS(status);
diffs[0] = 5.4;
ratios[0] = 5.4;
diffs[1] = 6.5;
ratios[1] = 6.5;
diffs[2] = 7.5;
ratios[2] = 7.5;
status = lefwLayerAntennaCumDiffAreaRatioPwl(3, diffs, ratios);
CHECK_STATUS(status);
status = lefwLayerAntennaCumAreaRatio(6.7);
CHECK_STATUS(status);
status = lefwLayerAntennaModel("OXIDE2");
CHECK_STATUS(status);
status = lefwLayerAntennaCumAreaRatio(300);
CHECK_STATUS(status);
status = lefwLayerAntennaCumRoutingPlusCut();            // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaAreaMinusDiff(100.0);           // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaGatePlusDiff(2.0);              // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaDiffAreaRatio(1000);            // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaCumDiffAreaRatio(5000);         // 5.7
CHECK_STATUS(status);
```

```
xl = (double*)malloc(sizeof(double)*5);
yl = (double*)malloc(sizeof(double)*5);
xl[0] = 0.0;
yl[0] = 1.0;
xl[1] = 0.09999;
yl[1] = 1.0;
xl[2] = 0.1;
yl[2] = 0.2;
xl[3] = 1.0;
yl[3] = 0.1;
xl[4] = 100;
yl[4] = 0.1;
status = lefwLayerAntennaAreaDiffReducePwl(5, xl, yl);   // 5.7
CHECK_STATUS(status);
free((char*)xl);
free((char*)yl);
diffs[0] = 1;
ratios[0] = 4;
diffs[1] = 2;
ratios[1] = 5;
status = lefwLayerAntennaCumDiffAreaRatioPwl(2, diffs, ratios);
CHECK_STATUS(status);
status = lefwLayerACCurrentDensity("PEAK", 0);
CHECK_STATUS(status);
current[0] = 1E6;
current[1] = 100E6;
status = lefwLayerACFrequency(2, current);
CHECK_STATUS(status);
current[0] = 0.5E-6;
current[1] = 0.4E-6;
status = lefwLayerACTableEntries(2, current);
CHECK_STATUS(status);
status = lefwLayerACCurrentDensity("AVERAGE", 0);
CHECK_STATUS(status);
current[0] = 1E6;
current[1] = 100E6;
status = lefwLayerACFrequency(2, current);
CHECK_STATUS(status);
current[0] = 0.6E-6;
current[1] = 0.5E-6;
status = lefwLayerACTableEntries(2, current);
```

```
CHECK_STATUS(status);
status = lefwLayerACCurrentDensity("RMS", 0);
CHECK_STATUS(status);
current[0] = 100E6;
current[1] = 800E6;
status = lefwLayerACFrequency(2, current);
CHECK_STATUS(status);
current[0] = 0.5E-6;
current[1] = 0.4E-6;
status = lefwLayerACTableEntries(2, current);
CHECK_STATUS(status);
status = lefwEndLayer("CUT12");
CHECK_STATUS(status);

status = lefwStartLayerRouting("PC");
CHECK_STATUS(status);
status = lefwLayerRouting("DIAG45", 1);
CHECK_STATUS(status);
status = lefwLayerRoutingPitch(1.8);
CHECK_STATUS(status);
status = lefwLayerRoutingWireExtension(0.4);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.6);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(1.2);                     // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingEndOfLine(1.3, 0.6);      // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(1.3);                     // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingEndOfLine(1.4, 0.7);      // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingEOLParallel(1.1, 0.5, 1); // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(1.4);                     // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingEndOfLine(1.5, 0.8);      // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingEOLParallel(1.2, 0.6, 0); // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingOffsetXYDistance(0.9, 0.7);
```

```
CHECK_STATUS(status);
status = lefwLayerRoutingResistance("PWL ( ( 1 0.103 ) )");
CHECK_STATUS(status);
status = lefwLayerRoutingCapacitance("PWL ( ( 1 0.000156 ) ( 10 0.001 ) )");
CHECK_STATUS(status);
status = lefwLayerAntennaAreaRatio(5.4);
CHECK_STATUS(status);
status = lefwLayerAntennaDiffAreaRatio(6.5);
CHECK_STATUS(status);
diffs[0] = 4.0;
ratios[0] = 4.1;
diffs[1] = 4.2;
ratios[1] = 4.3;
status = lefwLayerAntennaDiffAreaRatioPwl(2, diffs, ratios);
CHECK_STATUS(status);
status = lefwLayerAntennaCumAreaRatio(7.5);
CHECK_STATUS(status);
diffs[0] = 5.0;
ratios[0] = 5.1;
diffs[1] = 6.0;
ratios[1] = 6.1;
status = lefwLayerAntennaCumDiffAreaRatioPwl(2, diffs, ratios);
CHECK_STATUS(status);
status = lefwLayerAntennaAreaFactor(4.5, 0);
CHECK_STATUS(status);
status = lefwLayerAntennaSideAreaRatio(6.5);
CHECK_STATUS(status);
status = lefwLayerAntennaCumDiffSideAreaRatio(4.6);
CHECK_STATUS(status);
diffs[0] = 8.0;
ratios[0] = 8.1;
diffs[1] = 8.2;
ratios[1] = 8.3;
diffs[2] = 8.4;
ratios[2] = 8.5;
diffs[3] = 8.6;
ratios[3] = 8.7;
status = lefwLayerAntennaCumDiffSideAreaRatioPwl(4, diffs, ratios);
CHECK_STATUS(status);
status = lefwLayerAntennaCumSideAreaRatio(7.4);
CHECK_STATUS(status);
```

```
diffs[0] = 7.0;
ratios[0] = 7.1;
diffs[1] = 7.2;
ratios[1] = 7.3;
status = lefwLayerAntennaDiffSideAreaRatioPwl(2, diffs, ratios);
CHECK_STATUS(status);
status = lefwLayerAntennaSideAreaFactor(9.0, "DIFFUSEONLY");
CHECK_STATUS(status);
status = lefwLayerDCCurrentDensity("AVERAGE", 0);
CHECK_STATUS(status);
current[0] = 20.0;
current[1] = 50.0;
current[2] = 100.0;
status = lefwLayerDCWidth(3, current);
CHECK_STATUS(status);
current[0] = 1.0E-6;
current[1] = 0.7E-6;
current[2] = 0.5E-6;
status = lefwLayerDCTableEntries(3, current);
CHECK_STATUS(status);
status = lefwEndLayerRouting("PC");
CHECK_STATUS(status);

status = lefwStartLayer("CA", "CUT");
CHECK_STATUS(status);
status = lefwLayerCutSpacing(0.15);                // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingCenterToCenter();      // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                 // 5.7
CHECK_STATUS(status);
status = lefwLayerEnclosure("BELOW", 0.3, 0.01, 0);
CHECK_STATUS(status);
status = lefwLayerEnclosure("ABOVE", 0.5, 0.01, 0);
CHECK_STATUS(status);
status = lefwLayerPreferEnclosure("BELOW", 0.06, 0.01, 0);
CHECK_STATUS(status);
status = lefwLayerPreferEnclosure("ABOVE", 0.08, 0.02, 0);
CHECK_STATUS(status);
status = lefwLayerEnclosure("", 0.02, 0.02, 1.0);
CHECK_STATUS(status);
```

```
status = lefwLayerEnclosure(NULL, 0.05, 0.05, 2.0);
CHECK_STATUS(status);
status = lefwLayerEnclosure("BELOW", 0.07, 0.07, 1.0);
CHECK_STATUS(status);
status = lefwLayerEnclosure("ABOVE", 0.09, 0.09, 1.0);
CHECK_STATUS(status);
status = lefwLayerResistancePerCut(10.0);
CHECK_STATUS(status);
status = lefwLayerDCCurrentDensity("AVERAGE", 0);
CHECK_STATUS(status);
current[0] = 2.0;
current[1] = 5.0;
current[2] = 10.0;
status = lefwLayerDCWidth(3, current);
CHECK_STATUS(status);
current[0] = 0.6E-6;
current[1] = 0.5E-6;
current[2] = 0.4E-6;
status = lefwLayerDCTableEntries(3, current);
CHECK_STATUS(status);
status = lefwEndLayer("CA");
CHECK_STATUS(status);

status = lefwStartLayerRouting("M1");
CHECK_STATUS(status);
status = lefwLayerRouting("DIAG135", 1);
CHECK_STATUS(status);
status = lefwLayerRoutingPitch(1.8);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.6);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingRange(1.1, 100.1);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingRangeUseLengthThreshold();
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.61);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingRange(1.1, 100.1);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingRangeInfluence(2.01, 2.0, 1000.0);
CHECK_STATUS(status);
```

```
status = lefwLayerRoutingSpacing(0.62);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingRange(1.1, 100.1);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingRangeRange(4.1, 6.5);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.63);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingLengthThreshold(1.34, 4.5, 6.5);
CHECK_STATUS(status);
status = lefwLayerRoutingWireExtension(7);
CHECK_STATUS(status);
status = lefwLayerRoutingResistance("0.103");
CHECK_STATUS(status);
status = lefwLayerRoutingCapacitance("0.000156");
CHECK_STATUS(status);
current[0] = 0.00;
current[1] = 0.50;
current[2] = 3.00;
current[3] = 5.00;
status = lefwLayerRoutingStartSpacingtableParallel(4, current);
CHECK_STATUS(status);
current[0] = 0.15;
current[1] = 0.15;
current[2] = 0.15;
current[3] = 0.15;
status = lefwLayerRoutingSpacingtableParallelWidth(0.00, 4, current);
CHECK_STATUS(status);
current[0] = 0.15;
current[1] = 0.20;
current[2] = 0.20;
current[3] = 0.20;
status = lefwLayerRoutingSpacingtableParallelWidth(0.25, 4, current);
CHECK_STATUS(status);
current[0] = 0.15;
current[1] = 0.50;
current[2] = 0.50;
current[3] = 0.50;
status = lefwLayerRoutingSpacingtableParallelWidth(1.50, 4, current);
CHECK_STATUS(status);
current[0] = 0.15;
```

```
current[1] = 0.50;
current[2] = 1.00;
current[3] = 1.00;
status = lefwLayerRoutingSpacingtableParallelWidth(3.00, 4, current);
CHECK_STATUS(status);
current[0] = 0.15;
current[1] = 0.50;
current[2] = 1.00;
current[3] = 2.00;
status = lefwLayerRoutingSpacingtableParallelWidth(5.00, 4, current);
CHECK_STATUS(status);
status = lefwLayerRoutineEndSpacingtable();
CHECK_STATUS(status);
status = lefwLayerRoutingStartSpacingtableInfluence();
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingInfluenceWidth(1.5, 0.5, 0.5);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingInfluenceWidth(3.0, 1.0, 1.0);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingInfluenceWidth(5.0, 2.0, 2.0);
CHECK_STATUS(status);
status = lefwLayerRoutineEndSpacingtable();
CHECK_STATUS(status);
status = lefwLayerRoutingStartSpacingtableInfluence();
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingInfluenceWidth(1.5, 0.5, 0.5);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingInfluenceWidth(5.0, 2.0, 2.0);
CHECK_STATUS(status);
status = lefwLayerRoutineEndSpacingtable();
CHECK_STATUS(status);
current[0] = 0.00;
current[1] = 0.50;
current[2] = 5.00;
status = lefwLayerRoutingStartSpacingtableParallel(3, current);
CHECK_STATUS(status);
current[0] = 0.15;
current[1] = 0.15;
current[2] = 0.15;
status = lefwLayerRoutingSpacingtableParallelWidth(0.00, 3, current);
CHECK_STATUS(status);
```

```
current[0] = 0.15;
current[1] = 0.20;
current[2] = 0.20;
status = lefwLayerRoutingSpacingtableParallelWidth(0.25, 3, current);
CHECK_STATUS(status);
current[0] = 0.15;
current[1] = 0.50;
current[2] = 1.00;
status = lefwLayerRoutingSpacingtableParallelWidth(3.00, 3, current);
CHECK_STATUS(status);
current[0] = 0.15;
current[1] = 0.50;
current[2] = 2.00;
status = lefwLayerRoutingSpacingtableParallelWidth(5.00, 3, current);
CHECK_STATUS(status);
status = lefwLayerRoutineEndSpacingtable();
CHECK_STATUS(status);
free((char*)current);
free((char*)diffs);
free((char*)ratios);
status = lefwLayerAntennaGatePlusDiff(2.0);      // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaDiffAreaRatio(1000);    // 5.7
CHECK_STATUS(status);
status = lefwLayerAntennaCumDiffAreaRatio(5000); // 5.7
CHECK_STATUS(status);
status = lefwEndLayerRouting("M1");
CHECK_STATUS(status);

status = lefwStartLayer("V1", "CUT");
CHECK_STATUS(status);
status = lefwLayerCutSpacing(0.6);
CHECK_STATUS(status);
status = lefwLayerCutSpacingLayer("CA", 0);
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();
CHECK_STATUS(status);
status = lefwEndLayer("V1");
CHECK_STATUS(status);

status = lefwStartLayerRouting("M2");
```

```
CHECK_STATUS(status);
status = lefwLayerRouting("VERTICAL", 0.9);
CHECK_STATUS(status);
status = lefwLayerRoutingPitch(1.8);
CHECK_STATUS(status);
status = lefwLayerRoutingWireExtension(8);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.9);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingLengthThreshold(100.9, 0, 0);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.5);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingLengthThreshold(0.9, 0, 0.1);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.6);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingLengthThreshold(1.9, 0, 0);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(1.0);              // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingSameNet(1);         // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(1.1);              // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingSameNet(0);         // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingResistance("0.0608");
CHECK_STATUS(status);
status = lefwLayerRoutingCapacitance("0.000184");
CHECK_STATUS(status);
status = lefwEndLayerRouting("M2");
CHECK_STATUS(status);

status = lefwStartLayer("V2", "CUT");
CHECK_STATUS(status);
status = lefwEndLayer("V2");
CHECK_STATUS(status);

status = lefwStartLayerRouting("M3");
CHECK_STATUS(status);
```

```
status = lefwLayerRouting("HORIZONTAL", 0.9);
CHECK_STATUS(status);
status = lefwLayerRoutingPitchXYDistance(1.8, 1.5);
CHECK_STATUS(status);
status = lefwLayerRoutingDiagPitchXYDistance(1.5, 1.8);
CHECK_STATUS(status);
status = lefwLayerRoutingWireExtension(8);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.9);
CHECK_STATUS(status);
status = lefwLayerRoutingResistance("0.0608");
CHECK_STATUS(status);
status = lefwLayerRoutingCapacitance("0.000184");
CHECK_STATUS(status);
status = lefwEndLayerRouting("M3");
CHECK_STATUS(status);

area = (double*)malloc(sizeof(double)*3);
width = (double*)malloc(sizeof(double)*3);

status = lefwStartLayerRouting("M4");
CHECK_STATUS(status);
status = lefwLayerRouting("HORIZONTAL", 0.9);
CHECK_STATUS(status);
status = lefwLayerRoutingMinimumcut(2, 0.50);
CHECK_STATUS(status);
status = lefwLayerRoutingMinimumcut(2, 0.70);
CHECK_STATUS(status);
status = lefwLayerRoutingMinimumcutConnections("FROMBELOW");
CHECK_STATUS(status);
status = lefwLayerRoutingMinimumcut(4, 1.0);
CHECK_STATUS(status);
status = lefwLayerRoutingMinimumcutConnections("FROMABOVE");
CHECK_STATUS(status);
status = lefwLayerRoutingMinimumcut(2, 1.1);
CHECK_STATUS(status);
status = lefwLayerRoutingMinimumcutLengthWithin(20.0, 5.0);
CHECK_STATUS(status);
area[0]  = 0.40;
width[0] = 0;
area[1]  = 0.40;
```

```
width[1] = 0.15;
area[2]  = 0.80;
width[2] = 0.50;
status = lefwLayerRoutingMinenclosedarea(3, area, width);
CHECK_STATUS(status);
status = lefwLayerRoutingMaxwidth(10.0);
CHECK_STATUS(status);
status = lefwLayerRoutingProtrusion(0.30, 0.60, 1.20);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstep(0.20);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstep(0.05);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstepWithOptions(0.05, NULL, 0.08);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstepWithOptions(0.05, NULL, 0.16);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstepWithOptions(0.05, "INSDECORNER", 0);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstepWithOptions(0.05, "INSIDECORNER", 0.15);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstepWithOptions(0.05, "STEP", 0);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstepWithOptions(0.05, "STEP", 0.08);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstepWithOptions(0.04, "STEP", 0);
CHECK_STATUS(status);
status = lefwLayerRoutingMinstepMaxEdges(1.0, 2);    // 5.7
CHECK_STATUS(status);
status = lefwEndLayerRouting("M4");
CHECK_STATUS(status);
free((char*)area);
free((char*)width);

status = lefwStartLayer("implant1", "IMPLANT");
CHECK_STATUS(status);
status = lefwLayerWidth(0.50);
CHECK_STATUS(status);
status = lefwLayerCutSpacing(0.50);
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();
```

```
CHECK_STATUS(status);
status = lefwEndLayer("implant1");
CHECK_STATUS(status);

status = lefwStartLayer("implant2", "IMPLANT");
CHECK_STATUS(status);
status = lefwLayerWidth(0.50);
CHECK_STATUS(status);
status = lefwLayerCutSpacing(0.50);
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();
CHECK_STATUS(status);
status = lefwEndLayer("implant2");
CHECK_STATUS(status);

status = lefwStartLayer("V3", "CUT");
CHECK_STATUS(status);
status = lefwLayerWidth(0.60);
CHECK_STATUS(status);
status = lefwEndLayer("V3");
CHECK_STATUS(status);

status = lefwStartLayerRouting("MT");
CHECK_STATUS(status);
status = lefwLayerRouting("VERTICAL", 0.9);
CHECK_STATUS(status);
status = lefwLayerRoutingPitch(1.8);
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.9);
CHECK_STATUS(status);
status = lefwLayerRoutingResistance("0.0608");
CHECK_STATUS(status);
status = lefwLayerRoutingCapacitance("0.000184");
CHECK_STATUS(status);
status = lefwEndLayerRouting("MT");
CHECK_STATUS(status);

status = lefwStartLayer("OVERLAP", "OVERLAP");
CHECK_STATUS(status);
status = lefwEndLayer("OVERLAP");
CHECK_STATUS(status);
```

```
status = lefwStartLayerRouting("MET2");
CHECK_STATUS(status);
status = lefwLayerRouting("VERTICAL", 0.9);
CHECK_STATUS(status);
status = lefwMinimumDensity(20.2);
CHECK_STATUS(status);
status = lefwMaximumDensity(80.0);
CHECK_STATUS(status);
status = lefwDensityCheckWindow(200.0, 200.0);
CHECK_STATUS(status);
status = lefwDensityCheckStep(100.0);
CHECK_STATUS(status);
status = lefwFillActiveSpacing(3.0);
CHECK_STATUS(status);
status = lefwEndLayerRouting("MET2");
CHECK_STATUS(status);

status = lefwStartLayer("via34", "CUT");                  // 5.7
CHECK_STATUS(status);
status = lefwLayerWidth(0.25);                            // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacing(0.1);                        // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingCenterToCenter();             // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                        // 5.7
CHECK_STATUS(status);
status = lefwLayerEnclosure(0, .05, .01, 0);              // 5.7
CHECK_STATUS(status);
status = lefwLayerEnclosureLength(0, .05, 0, 0.7);        // 5.7
CHECK_STATUS(status);
status = lefwLayerEnclosure("BELOW", .07, .07, 1.0);      // 5.7
CHECK_STATUS(status);
status = lefwLayerEnclosure("ABOVE", .09, .09, 1.0);      // 5.7
CHECK_STATUS(status);
status = lefwLayerEnclosureWidth(0, .03, .03, 1.0, 0.2); // 5.7
CHECK_STATUS(status);
status = lefwEndLayer("via34");                           // 5.7
CHECK_STATUS(status);
```

```
status = lefwStartLayer("cut23", "CUT");                   // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacing(0.20);                        // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingSameNet();                     // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingLayer("cut12", 1);             // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                         // 5.7
CHECK_STATUS(status);

status = lefwLayerCutSpacing(0.30);                        // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingCenterToCenter();              // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingSameNet();                     // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingArea(0.02);                    // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                         // 5.7
CHECK_STATUS(status);

status = lefwLayerCutSpacing(0.40);                        // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingArea(0.5);                     // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                         // 5.7
CHECK_STATUS(status);

status = lefwLayerCutSpacing(0.10);                        // 5.7
CHECK_STATUS(status);
status = lefwLayerCutSpacingEnd();                         // 5.7
CHECK_STATUS(status);

wthn = (double*)malloc(sizeof(double)*3);                  // 5.7
spng = (double*)malloc(sizeof(double)*3);
wthn[0] = 0.15;
spng[0] = 0.11;
wthn[1] = 0.13;
spng[1] = 0.13;
wthn[2] = 0.11;
```

```
spng[2] = 0.15;
status = lefwLayerCutSpacingTableOrtho(3, wthn, spng);
CHECK_STATUS(status);

wthn[0] = 3;
spng[0] = 1;
status = lefwLayerArraySpacing(0, 2.0, 0.2, 1, wthn, spng);
CHECK_STATUS(status);
wthn[0] = 3;
spng[0] = 1;
wthn[1] = 4;
spng[1] = 1.5;
wthn[2] = 5;
spng[2] = 2.0;
status = lefwLayerArraySpacing(1, 2.0, 0.2, 3, wthn, spng);
CHECK_STATUS(status);
free((char*)wthn);
free((char*)spng);
status = lefwEndLayer("cut23");
CHECK_STATUS(status);

status = lefwStartLayerRouting("cut24");                  // 5.7
CHECK_STATUS(status);
status = lefwLayerRouting("HORIZONTAL", 1);               // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingPitch(1.2);                      // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.10);                   // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.12);                   // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingNotchLength(0.15);        // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacing(0.14);                   // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingSpacingEndOfNotchWidth(0.15, 0.16, 0.08); // 5.7
CHECK_STATUS(status);
status = lefwEndLayerRouting("cut24");                    // 5.7
CHECK_STATUS(status);

status = lefwStartLayerRouting("cut25");                  // 5.7
```

```
CHECK_STATUS(status);
status = lefwLayerRoutingPitch(1.2);                      // 5.7
CHECK_STATUS(status);
status = lefwLayerRouting("HORIZONTAL", 1);               // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingWireExtension(7);                // 5.7
CHECK_STATUS(status);
status = lefwLayerRoutingStartSpacingtableTwoWidths();    // 5.7
CHECK_STATUS(status);
wthn = (double*)malloc(sizeof(double)*4);                 // 5.7
wthn[0] = 0.15;
wthn[1] = 0.20;
wthn[2] = 0.50;
wthn[3] = 1.00;
status = lefwLayerRoutingSpacingtableTwoWidthsWidth(0.0, 0, 4, wthn);  // 5.7
CHECK_STATUS(status);
wthn[0] = 0.20;
wthn[1] = 0.25;
wthn[2] = 0.50;
wthn[3] = 1.00;
status = lefwLayerRoutingSpacingtableTwoWidthsWidth(0.25, 0.1, 4, wthn);// 5.7
CHECK_STATUS(status);
wthn[0] = 0.50;
wthn[1] = 0.50;
wthn[2] = 0.60;
wthn[3] = 1.00;
status = lefwLayerRoutingSpacingtableTwoWidthsWidth(1.5, 1.5, 4, wthn);// 5.7
CHECK_STATUS(status);
wthn[0] = 1.00;
wthn[1] = 1.00;
wthn[2] = 1.00;
wthn[3] = 1.20;
status = lefwLayerRoutingSpacingtableTwoWidthsWidth(3.0, 3.0, 4, wthn);// 5.7
CHECK_STATUS(status);
free(wthn);
status = lefwLayerRoutineEndSpacingtable();
CHECK_STATUS(status);
status = lefwEndLayerRouting("cut25");                    // 5.7
CHECK_STATUS(status);

// MAXVIASTACK
```

```
status = lefwMaxviastack(4, "m1", "m7");
CHECK_STATUS(status);

// VIA
status = lefwStartVia("RX_PC", "DEFAULT");
CHECK_STATUS(status);
status = lefwViaResistance(2);
CHECK_STATUS(status);
status = lefwViaLayer("RX");
CHECK_STATUS(status);
status = lefwViaLayerRect(-0.7, -0.7, 0.7, 0.7);
CHECK_STATUS(status);
status = lefwViaLayer("CUT12");
CHECK_STATUS(status);
status = lefwViaLayerRect(-0.25, -0.25, 0.25, 0.25);
CHECK_STATUS(status);
status = lefwViaLayer("PC");
CHECK_STATUS(status);
status = lefwViaLayerRect(-0.6, -0.6, 0.6, 0.6);
CHECK_STATUS(status);
status = lefwStringProperty("stringProperty", "DEFAULT");
CHECK_STATUS(status);
status = lefwRealProperty("realProperty", 32.33);
CHECK_STATUS(status);
status = lefwIntProperty("COUNT", 34);
CHECK_STATUS(status);
status = lefwEndVia("PC");
CHECK_STATUS(status);

status = lefwStartVia("M2_M3_PWR", NULL);
CHECK_STATUS(status);
status = lefwViaResistance(0.4);
CHECK_STATUS(status);
status = lefwViaLayer("M2");
CHECK_STATUS(status);
status = lefwViaLayerRect(-1.35, -1.35, 1.35, 1.35);
CHECK_STATUS(status);
status = lefwViaLayer("V2");
CHECK_STATUS(status);
status = lefwViaLayerRect(-1.35, -1.35, -0.45, 1.35);
CHECK_STATUS(status);
```

```
status = lefwViaLayerRect(0.45, -1.35, 1.35, -0.45);
CHECK_STATUS(status);
status = lefwViaLayerRect(0.45, 0.45, 1.35, 1.35);
CHECK_STATUS(status);
status = lefwViaLayer("M3");
CHECK_STATUS(status);
status = lefwViaLayerRect(-1.35, -1.35, 1.35, 1.35);
CHECK_STATUS(status);
status = lefwEndVia("M2_M3_PWR");
CHECK_STATUS(status);

xl = (double*)malloc(sizeof(double)*6);
yl = (double*)malloc(sizeof(double)*6);
status = lefwStartVia("IN1X", 0);
CHECK_STATUS(status);
status = lefwViaLayer("metal2");
CHECK_STATUS(status);
xl[0] = -2.1;
yl[0] = -1.0;
xl[1] = -0.2;
yl[1] = 1.0;
xl[2] = 2.1;
yl[2] = 1.0;
xl[3] = 0.2;
yl[3] = -1.0;
xl[4] = 0.2;
yl[4] = -1.0;
xl[5] = 0.2;
yl[5] = -1.0;
status = lefwViaLayerPolygon(6, xl, yl);
CHECK_STATUS(status);
xl[0] = -1.1;
yl[0] = -2.0;
xl[1] = -0.1;
yl[1] = 2.0;
xl[2] = 1.1;
yl[2] = 2.0;
xl[3] = 0.1;
yl[3] = -2.0;
status = lefwViaLayerPolygon(4, xl, yl);
CHECK_STATUS(status);
```

```
xl[0] = -3.1;
yl[0] = -2.0;
xl[1] = -0.3;
yl[1] = 2.0;
xl[2] = 3.1;
yl[2] = 2.0;
xl[3] = 0.3;
yl[3] = -2.0;
status = lefwViaLayerPolygon(4, xl, yl);
CHECK_STATUS(status);
xl[0] = -4.1;
yl[0] = -2.0;
xl[1] = -0.4;
yl[1] = 2.0;
xl[2] = 4.1;
yl[2] = 2.0;
xl[3] = 0.4;
yl[3] = -2.0;
status = lefwViaLayerPolygon(4, xl, yl);
CHECK_STATUS(status);
status = lefwViaLayer("cut23");
CHECK_STATUS(status);
status = lefwViaLayerRect(-0.4, -0.4, 0.4, 0.4);
CHECK_STATUS(status);
xl[0] = -2.1;
yl[0] = -1.0;
xl[1] = -0.2;
yl[1] = 1.0;
xl[2] = 2.1;
yl[2] = 1.0;
xl[3] = 0.2;
yl[3] = -1.0;
status = lefwViaLayerPolygon(4, xl, yl);
CHECK_STATUS(status);
status = lefwEndVia("IN1X");
CHECK_STATUS(status);

status = lefwStartVia("myBlockVia", NULL);
CHECK_STATUS(status);
status = lefwViaViarule("DEFAULT", 0.1, 0.1, "metal1", "via12", "metal2",
                        0.1, 0.1, 0.05, 0.01, 0.01, 0.05);
```

```
CHECK_STATUS(status);
status = lefwViaViaruleRowCol(1, 2);
CHECK_STATUS(status);
status = lefwViaViaruleOrigin(1.5, 2.5);
CHECK_STATUS(status);
status = lefwViaViaruleOffset(1.5, 2.5, 3.5, 4.5);
CHECK_STATUS(status);
status = lefwViaViarulePattern("2_1RF1RF1R71R0_3_R1FFFF");
CHECK_STATUS(status);
status = lefwEndVia("myBlockVia");
CHECK_STATUS(status);

status = lefwStartVia("myVia23", NULL);
CHECK_STATUS(status);
status = lefwViaLayer("metal2");
CHECK_STATUS(status);
status = lefwViaLayerPolygon(6, xl, yl);
CHECK_STATUS(status);
status = lefwViaLayer("cut23");
CHECK_STATUS(status);
status = lefwViaLayerRect(-0.4, -0.4, 0.4, 0.4);
CHECK_STATUS(status);
status = lefwViaLayer("metal3");
CHECK_STATUS(status);
status = lefwViaLayerPolygon(5, xl, yl);
CHECK_STATUS(status);
status = lefwEndVia("myVia23");
CHECK_STATUS(status);

free((char*)xl);
free((char*)yl);

// VIARULE
status = lefwStartViaRule("VIALIST12");
CHECK_STATUS(status);
lefwAddComment("Break up the old lefwViaRule into 2 routines");
lefwAddComment("lefwViaRuleLayer and lefwViaRuleVia");
status = lefwViaRuleLayer("M1", NULL, 9.0, 9.6, 0, 0);
CHECK_STATUS(status);
status = lefwViaRuleLayer("M2", NULL, 3.0, 3.0, 0, 0);
CHECK_STATUS(status);
```

```
status = lefwViaRuleVia("VIACENTER12");
CHECK_STATUS(status);
status = lefwStringProperty("vrsp", "new");
CHECK_STATUS(status);
status = lefwIntProperty("vrip", 1);
CHECK_STATUS(status);
status = lefwRealProperty("vrrp", 4.5);
CHECK_STATUS(status);
status = lefwEndViaRule("VIALIST12");
CHECK_STATUS(status);

// VIARULE with GENERATE
lefwAddComment("Break up the old lefwViaRuleGenearte into 4 routines");
lefwAddComment("lefwStartViaRuleGen, lefwViaRuleGenLayer,");
lefwAddComment("lefwViaRuleGenLayer3, and lefwEndViaRuleGen");
status = lefwStartViaRuleGen("VIAGEN12");
CHECK_STATUS(status);
status = lefwViaRuleGenLayer("M1", NULL, 0.1, 19, 0, 0);
CHECK_STATUS(status);
status = lefwViaRuleGenLayer("M2", NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwViaRuleGenLayer3("V1", -0.8, -0.8, 0.8, 0.8, 5.6, 6.0, 0.2);
CHECK_STATUS(status);
status = lefwEndViaRuleGen("VIAGEN12");
CHECK_STATUS(status);

// VIARULE with GENERATE & ENCLOSURE & DEFAULT
status = lefwStartViaRuleGen("via12");
CHECK_STATUS(status);
status = lefwViaRuleGenDefault();
CHECK_STATUS(status);
status = lefwViaRuleGenLayerEnclosure("m1", 0.05, 0.005, 1.0, 100.0);
CHECK_STATUS(status);
status = lefwViaRuleGenLayerEnclosure("m2", 0.05, 0.005, 1.0, 100.0);
CHECK_STATUS(status);
status = lefwViaRuleGenLayer3("cut12", -0.07, -0.07, 0.07, 0.07, 0.16, 0.16, 0);
CHECK_STATUS(status);
status = lefwEndViaRuleGen("via12");
CHECK_STATUS(status);

// NONDEFAULTRULE
```

```
status = lefwStartNonDefaultRule("RULE1");
CHECK_STATUS(status);
status = lefwNonDefaultRuleHardspacing();
CHECK_STATUS(status);
status = lefwNonDefaultRuleLayer("RX", 10.0, 2.2, 6, 0, 0, 0);
CHECK_STATUS(status);
status = lefwNonDefaultRuleLayer("PC", 10.0, 2.2, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwNonDefaultRuleLayer("M1", 10.0, 2.2, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwStartVia("nd1VARX0", NULL);
CHECK_STATUS(status);
status = lefwViaResistance(0.2);
CHECK_STATUS(status);
status = lefwViaLayer("RX");
CHECK_STATUS(status);
status = lefwViaLayerRect(-3, -3, 3, 3);
CHECK_STATUS(status);
status = lefwViaLayer("CUT12");
CHECK_STATUS(status);
status = lefwViaLayerRect(-1.0, -1.0, 1.0, 1.0);
CHECK_STATUS(status);
status = lefwViaLayer("PC");
CHECK_STATUS(status);
status = lefwViaLayerRect(-3, -3, 3, 3);
CHECK_STATUS(status);
status = lefwEndVia("nd1VARX0");
CHECK_STATUS(status);
status = lefwStartSpacing();
CHECK_STATUS(status);
status = lefwSpacing("CUT01", "RX", 0.1, "STACK");
CHECK_STATUS(status);
status = lefwEndSpacing();
CHECK_STATUS(status);
status = lefwEndNonDefaultRule("RULE1");
CHECK_STATUS(status);
status = lefwStartNonDefaultRule("wide1_5x");
CHECK_STATUS(status);
status = lefwNonDefaultRuleLayer("fw", 4.8, 4.8, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwNonDefaultRuleStartVia("nd1VIARX0", "DEFAULT");
```

```
CHECK_STATUS(status);
status = lefwViaResistance(0.2);
CHECK_STATUS(status);
status = lefwViaLayer("RX");
CHECK_STATUS(status);
status = lefwViaLayerRect(-3, -3, 3, 3);
CHECK_STATUS(status);
status = lefwViaLayer("CUT12");
CHECK_STATUS(status);
status = lefwViaLayerRect(-1.0, -1.0, 1.0, 1.0);
CHECK_STATUS(status);
status = lefwViaLayer("PC");
CHECK_STATUS(status);
status = lefwViaLayerRect(-3, -3, 3, 3);
CHECK_STATUS(status);
status = lefwNonDefaultRuleEndVia("nd1VIARX0");
CHECK_STATUS(status);
status = lefwNonDefaultRuleUseVia("via12_fixed_analog_via");
CHECK_STATUS(status);
status = lefwNonDefaultRuleMinCuts("cut12", 2);
CHECK_STATUS(status);
status = lefwNonDefaultRuleUseVia("via23_fixed_analog_via");
CHECK_STATUS(status);
status = lefwNonDefaultRuleMinCuts("cut23", 2);
CHECK_STATUS(status);
status = lefwNonDefaultRuleUseViaRule("viaRule23_fixed_analog_via");
CHECK_STATUS(status);
status = lefwEndNonDefaultRule("wide1_5x");
CHECK_STATUS(status);

// SPACING
status = lefwStartSpacing();
CHECK_STATUS(status);
status = lefwSpacing("CUT01", "CA", 1.5, NULL);
CHECK_STATUS(status);
status = lefwSpacing("CA", "V1", 1.5, "STACK");
CHECK_STATUS(status);
status = lefwSpacing("M1", "M1", 3.5, "STACK");
CHECK_STATUS(status);
status = lefwSpacing("V1", "V2", 1.5, "STACK");
CHECK_STATUS(status);
```

```
status = lefwSpacing("M2", "M2", 3.5, "STACK");
CHECK_STATUS(status);
status = lefwSpacing("V2", "V3", 1.5, "STACK");
CHECK_STATUS(status);
status = lefwEndSpacing();
CHECK_STATUS(status);

// MINFEATURE & DIELECTRIC
status = lefwMinFeature(0.1, 0.1);
CHECK_STATUS(status);
status = lefwNewLine();
CHECK_STATUS(status);

// SITE
status = lefwSite("CORE1", "CORE", "X", 67.2, 6);
CHECK_STATUS(status);
status = lefwSiteRowPattern("Fsite", 0);
CHECK_STATUS(status);
status = lefwSiteRowPatternStr("Lsite", "N");
CHECK_STATUS(status);
status = lefwSiteRowPatternStr("Lsite", "FS");
CHECK_STATUS(status);
lefwEndSite("CORE1");
CHECK_STATUS(status);
status = lefwSite("CORE", "CORE", "Y", 3.6, 28.8);
CHECK_STATUS(status);
lefwEndSite("CORE");
CHECK_STATUS(status);
status = lefwSite("MRCORE", "CORE", "Y", 3.6, 28.8);
CHECK_STATUS(status);
lefwEndSite("MRCORE");
CHECK_STATUS(status);
status = lefwSite("IOWIRED", "PAD", NULL, 57.6, 432);
CHECK_STATUS(status);
lefwEndSite("IOWIRED");
CHECK_STATUS(status);

// ARRAY
status = lefwStartArray("M7E4XXX");
CHECK_STATUS(status);
status = lefwArraySite("CORE", -5021.450, -4998.000, 0, 14346, 595, 0.700,
```

```
                        16.800);
  CHECK_STATUS(status);
  status = lefwArraySiteStr("CORE", -5021.450, -4998.600, "FS", 14346, 595,
                           0.700, 16.800);
  CHECK_STATUS(status);
  status = lefwArraySite("IO", 6148.800, 5800.000, 3, 1, 1, 0.000, 0.000);
  CHECK_STATUS(status);
  status = lefwArraySiteStr("IO", 6148.800, 5240.000, "E", 1, 1, 0.000, 0.000);
  CHECK_STATUS(status);
  status = lefwArraySite("COVER", -7315.0, -7315.000, 1, 1, 1, 0.000, 0.000);
  CHECK_STATUS(status);
  status = lefwArraySiteStr("COVER", 7315.0, 7315.000, "FN", 1, 1, 0.000, 0.000);
  CHECK_STATUS(status);
  status = lefwArrayCanplace("COVER", -7315.000, -7315.000, 0, 1, 1, 0.000,
                            0.000);
  CHECK_STATUS(status);
  status = lefwArrayCanplaceStr("COVER", -7250.000, -7250.000, "N", 5, 1,
                               40.000, 0.000);
  CHECK_STATUS(status);
  status = lefwArrayCannotoccupy("CORE", -5021.450, -4989.600, 6, 100, 595,
                                0.700, 16.800);
  CHECK_STATUS(status);
  status = lefwArrayCannotoccupyStr("CORE", -5021.450, -4989.600, "N", 100, 595,
                                0.700, 16.800);
  CHECK_STATUS(status);
  status = lefwArrayTracks("X", -6148.800, 17569, 0.700, "RX");
  CHECK_STATUS(status);
  status = lefwArrayTracks("Y", -6148.800, 20497, 0.600, "RX");
  CHECK_STATUS(status);
  status = lefwStartArrayFloorplan("100%");
  CHECK_STATUS(status);
  status = lefwArrayFloorplan("CANPLACE", "COVER", -7315.000, -7315.000, 1, 1,
                             1, 0.000, 0.000);
  CHECK_STATUS(status);
  status = lefwArrayFloorplanStr("CANPLACE", "COVER", -7250.000, -7250.000,
                                "N", 5, 1, 40.000, 0.000);
  CHECK_STATUS(status);
  status = lefwArrayFloorplan("CANPLACE", "CORE", -5021.000, -4998.000, 1,
                             14346, 595, 0.700, 16.800);
  CHECK_STATUS(status);
  status = lefwArrayFloorplanStr("CANPLACE", "CORE", -5021.000, -4998.000, "FS",
```

```
                              100, 595, 0.700, 16.800);
CHECK_STATUS(status);
status = lefwArrayFloorplan("CANNOTOCCUPY", "CORE", -5021.000, -4998.000, 7,
                           14346, 595, 0.700, 16.800);
CHECK_STATUS(status);
status = lefwArrayFloorplanStr("CANNOTOCCUPY", "CORE", -5021.000, -4998.000,
                              "E", 100, 595, 0.700, 16.800);
CHECK_STATUS(status);
status = lefwEndArrayFloorplan("100%");
CHECK_STATUS(status);
status = lefwArrayGcellgrid("X", -6157.200, 1467, 8.400);
CHECK_STATUS(status);
status = lefwArrayGcellgrid("Y", -6157.200, 1467, 8.400);
CHECK_STATUS(status);
status = lefwEndArray("M7E4XXX");
CHECK_STATUS(status);

// MACRO
status = lefwStartMacro("CHK3A");
CHECK_STATUS(status);
status = lefwMacroClass("RING", NULL);
CHECK_STATUS(status);
status = lefwMacroOrigin(0.9, 0.9);
CHECK_STATUS(status);
status = lefwMacroSize(10.8, 28.8);
CHECK_STATUS(status);
status = lefwMacroSymmetry("X Y R90");
CHECK_STATUS(status);
status = lefwMacroSite("CORE");
CHECK_STATUS(status);
status = lefwStartMacroPin("GND");
CHECK_STATUS(status);
status = lefwMacroPinDirection("INOUT");
CHECK_STATUS(status);
status = lefwMacroPinMustjoin("PA3");
CHECK_STATUS(status);
status = lefwMacroPinTaperRule("RULE1");
CHECK_STATUS(status);
status = lefwMacroPinUse("GROUND");
CHECK_STATUS(status);
status = lefwMacroPinShape("ABUTMENT");
```

```
CHECK_STATUS(status);
status = lefwMacroPinSupplySensitivity("vddpin1");
CHECK_STATUS(status);
status = lefwMacroPinNetExpr("power1 VDD1");
CHECK_STATUS(status);
status = lefwMacroPinAntennaMetalArea(3, "M1");
CHECK_STATUS(status);
// MACRO - PIN
status = lefwStartMacroPinPort(NULL);
CHECK_STATUS(status);
status = lefwMacroPinPortLayer("M1", 0.05);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerRect(-0.9, 3, 9.9, 6, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwEndMacroPinPort();
CHECK_STATUS(status);
status = lefwStringProperty("TYPE", "special");
CHECK_STATUS(status);
status = lefwIntProperty("intProp", 23);
CHECK_STATUS(status);
status = lefwRealProperty("realProp", 24.25);
CHECK_STATUS(status);
status = lefwMacroPinAntennaModel("OXIDE1");
CHECK_STATUS(status);
status = lefwEndMacroPin("GND");
CHECK_STATUS(status);
status = lefwStartMacroPin("VDD");
CHECK_STATUS(status);
status = lefwMacroPinDirection("INOUT");
CHECK_STATUS(status);
status = lefwMacroPinUse("POWER");
CHECK_STATUS(status);
status = lefwMacroPinShape("ABUTMENT");
CHECK_STATUS(status);
status = lefwMacroPinNetExpr("power2 VDD2");
CHECK_STATUS(status);
// MACRO - PIN - PORT
status = lefwStartMacroPinPort(NULL);
CHECK_STATUS(status);
status = lefwMacroPinPortLayer("M1", 0);
CHECK_STATUS(status);
```

```
status = lefwMacroPinPortLayerRect(-0.9, 21, 9.9, 24, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwMacroPinPortVia(100, 300, "nd1VIA12", 1, 2, 1, 2);
CHECK_STATUS(status);
status = lefwEndMacroPinPort();
CHECK_STATUS(status);
status = lefwStartMacroPinPort("BUMP");
CHECK_STATUS(status);
status = lefwMacroPinPortLayer("M2", 0.06);
CHECK_STATUS(status);
status = lefwEndMacroPinPort();
CHECK_STATUS(status);
xl = (double*)malloc(sizeof(double)*5);
yl = (double*)malloc(sizeof(double)*5);
xl[0] = 30.8;
yl[0] = 30.5;
xl[1] = 42;
yl[1] = 53.5;
xl[2] = 60.8;
yl[2] = 25.5;
xl[3] = 47;
yl[3] = 15.5;
xl[4] = 20.8;
yl[4] = 0.5;
status = lefwStartMacroPinPort("CORE");
CHECK_STATUS(status);
status = lefwMacroPinPortLayer("P1", 0);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerPolygon(5, xl, yl, 5, 6, 454.6, 345.6);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerPolygon(5, xl, yl, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwEndMacroPinPort();
CHECK_STATUS(status);
free((char*)xl);
free((char*)yl);
status = lefwEndMacroPin("VDD");
CHECK_STATUS(status);
status = lefwStartMacroPin("PA3");
CHECK_STATUS(status);
status = lefwMacroPinDirection("INPUT");
```

```
CHECK_STATUS(status);
status = lefwMacroPinNetExpr("gnd1 GND");
CHECK_STATUS(status);
// 5.4
status = lefwMacroPinAntennaPartialMetalArea(4, "M1");
CHECK_STATUS(status);
status = lefwMacroPinAntennaPartialMetalArea(5, "M2");
CHECK_STATUS(status);
status = lefwMacroPinAntennaPartialMetalSideArea(5, "M2");
CHECK_STATUS(status);
status = lefwMacroPinAntennaGateArea(1, "M1");
CHECK_STATUS(status);
status = lefwMacroPinAntennaGateArea(2, 0);
CHECK_STATUS(status);
status = lefwMacroPinAntennaGateArea(3, "M3");
CHECK_STATUS(status);
status = lefwMacroPinAntennaDiffArea(1, "M1");
CHECK_STATUS(status);
status = lefwMacroPinAntennaMaxAreaCar(1, "L1");
CHECK_STATUS(status);
status = lefwMacroPinAntennaMaxSideAreaCar(1, 0);
CHECK_STATUS(status);
status = lefwMacroPinAntennaPartialCutArea(1, 0);
CHECK_STATUS(status);
status = lefwMacroPinAntennaPartialCutArea(2, "M2");
CHECK_STATUS(status);
status = lefwMacroPinAntennaPartialCutArea(3, 0);
CHECK_STATUS(status);
status = lefwMacroPinAntennaPartialCutArea(4, "M4");
CHECK_STATUS(status);
status = lefwMacroPinAntennaMaxCutCar(1, 0);
CHECK_STATUS(status);
status = lefwStartMacroPinPort("CORE");
CHECK_STATUS(status);
status = lefwMacroPinPortLayer("M1", 0.02);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerRect(1.35, -0.45, 2.25, 0.45, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerRect(-0.45, -0.45, 0.45, 0.45, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwEndMacroPinPort();
```

```
CHECK_STATUS(status);
status = lefwStartMacroPinPort(NULL);
CHECK_STATUS(status);
status = lefwMacroPinPortLayer("PC", 0);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerRect(-0.45, 12.15, 0.45, 13.05, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwEndMacroPinPort();
CHECK_STATUS(status);
status = lefwStartMacroPinPort(NULL);
CHECK_STATUS(status);
status = lefwMacroPinPortDesignRuleWidth("PC", 2);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerRect(8.55, 8.55, 9.45, 9.45, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerRect(6.75, 6.75, 7.65, 7.65, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerRect(6.75, 8.75, 7.65, 9.65, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerRect(6.75, 10.35, 7.65, 11.25, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwEndMacroPinPort();
CHECK_STATUS(status);
status = lefwEndMacroPin("PA3");
CHECK_STATUS(status);
// MACRO - OBS
status = lefwStartMacroObs();
CHECK_STATUS(status);
status = lefwMacroObsLayer("M1", 5.6);
CHECK_STATUS(status);
status = lefwMacroObsLayerWidth(5.4);
CHECK_STATUS(status);
status = lefwMacroObsLayerRect(6.6, -0.6, 9.6, 0.6, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwMacroObsLayerRect(4.8, 12.9, 9.6, 13.2, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwMacroObsLayerRect(3, 13.8, 7.8, 16.8, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwMacroObsLayerRect(3, -0.6, 6, 0.6, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwEndMacroObs();
```

```
CHECK_STATUS(status);
status = lefwStringProperty("stringProp", "first");
CHECK_STATUS(status);
status = lefwIntProperty("integerProp", 1);
CHECK_STATUS(status);
status = lefwRealProperty("WEIGHT", 30.31);
CHECK_STATUS(status);
status = lefwEndMacro("CHK3A");
CHECK_STATUS(status);

// 2nd MACRO
status = lefwStartMacro("INV");
CHECK_STATUS(status);
status = lefwMacroEEQ("CHK1");
CHECK_STATUS(status);
status = lefwMacroClass("CORE", "SPACER");
CHECK_STATUS(status);
status = lefwMacroForeign("INVS", 0, 0, -1);
CHECK_STATUS(status);
status = lefwMacroSize(67.2, 24);
CHECK_STATUS(status);
status = lefwMacroSymmetry("X Y R90");
CHECK_STATUS(status);
status = lefwMacroSite("CORE1");
CHECK_STATUS(status);
status = lefwStartMacroDensity("metal1");
CHECK_STATUS(status);
status = lefwMacroDensityLayerRect(0, 0, 100, 100, 45.5);
CHECK_STATUS(status);
status = lefwMacroDensityLayerRect(100, 0, 200, 100, 42.2);
CHECK_STATUS(status);
status = lefwEndMacroDensity();
CHECK_STATUS(status);
status = lefwStartMacroDensity("metal2");
CHECK_STATUS(status);
status = lefwMacroDensityLayerRect(200, 1, 300, 200, 43.3);
CHECK_STATUS(status);
status = lefwEndMacroDensity();
CHECK_STATUS(status);
status = lefwStartMacroPin("Z");
CHECK_STATUS(status);
```

```
status = lefwMacroPinDirection("OUTPUT");
CHECK_STATUS(status);
status = lefwMacroPinUse("SIGNAL");
CHECK_STATUS(status);
status = lefwMacroPinShape("ABUTMENT");
CHECK_STATUS(status);
status = lefwMacroPinAntennaModel("OXIDE1");
CHECK_STATUS(status);
status = lefwStartMacroPinPort(NULL);
CHECK_STATUS(status);
status = lefwMacroPinPortLayer("M2", 0);
CHECK_STATUS(status);
status = lefwMacroPinPortLayerWidth(5.6);
CHECK_STATUS(status);
xpath = (double*)malloc(sizeof(double)*7);
ypath = (double*)malloc(sizeof(double)*7);
xpath[0] = 30.8;
ypath[0] = 9;
xpath[1] = 42;
ypath[1] = 9;
xpath[2] = 30.8;
ypath[2] = 9;
xpath[3] = 42;
ypath[3] = 9;
xpath[4] = 30.8;
ypath[4] = 9;
xpath[5] = 42;
ypath[5] = 9;
xpath[6] = 30.8;
ypath[6] = 9;
status = lefwMacroPinPortLayerPath(7, xpath, ypath, 0, 0, 0, 0);
CHECK_STATUS(status);
status = lefwEndMacroPinPort();
CHECK_STATUS(status);
status = lefwEndMacroPin("Z");
free((char*)xpath);
free((char*)ypath);
// MACRO - OBS
status = lefwStartMacroObs();
CHECK_STATUS(status);
status = lefwMacroObsDesignRuleWidth("M1", 2);
```

```
CHECK_STATUS(status);
status = lefwMacroObsLayerRect(24.1, 1.5, 43.5, 208.5, 0, 0, 0, 0);
CHECK_STATUS(status);
xpath = (double*)malloc(sizeof(double)*2);
ypath = (double*)malloc(sizeof(double)*2);
xpath[0] = 8.4;
ypath[0] = 3;
xpath[1] = 8.4;
ypath[1] = 124;
status = lefwMacroObsLayerPath(2, xpath, ypath, 0, 0, 0, 0);
CHECK_STATUS(status);
xpath[0] = 58.8;
ypath[0] = 3;
xpath[1] = 58.8;
ypath[1] = 123;
status = lefwMacroObsLayerPath(2, xpath, ypath, 0, 0, 0, 0);
CHECK_STATUS(status);
xpath[0] = 64.4;
ypath[0] = 3;
xpath[1] = 64.4;
ypath[1] = 123;
status = lefwMacroObsLayerPath(2, xpath, ypath, 0, 0, 0, 0);
CHECK_STATUS(status);
free((char*)xpath);
free((char*)ypath);
xl = (double*)malloc(sizeof(double)*5);
yl = (double*)malloc(sizeof(double)*5);
xl[0] = 6.4;
xl[1] = 3.4;
xl[2] = 5.4;
xl[3] = 8.4;
xl[4] = 9.4;
yl[0] = 9.2;
yl[1] = 0.2;
yl[2] = 7.2;
yl[3] = 8.2;
yl[4] = 1.2;
status = lefwMacroObsLayerPolygon(5, xl, yl, 0, 0, 0, 0);
CHECK_STATUS(status);
free((char*)xl);
free((char*)yl);
```

```
status = lefwEndMacroObs();
CHECK_STATUS(status);
status = lefwEndMacro("INV");
CHECK_STATUS(status);

// 3rd MACRO
status = lefwStartMacro("DFF3");
CHECK_STATUS(status);
status = lefwMacroClass("CORE", "ANTENNACELL");
CHECK_STATUS(status);
status = lefwMacroForeignStr("DFF3S", 0, 0, "N");
CHECK_STATUS(status);
status = lefwMacroSize(67.2, 210);
CHECK_STATUS(status);
status = lefwMacroSymmetry("X Y R90");
CHECK_STATUS(status);
status = lefwMacroSitePattern("CORE", 34, 54, 7, 30, 3, 1, 1);
CHECK_STATUS(status);
status = lefwMacroSitePatternStr("CORE1", 21, 68, "S", 30, 3, 2, 2);
CHECK_STATUS(status);
status = lefwEndMacro("DFF3");
CHECK_STATUS(status);

status = lefwStartMacro("DFF4");
CHECK_STATUS(status);
status = lefwMacroClass("COVER", "BUMP");
CHECK_STATUS(status);
status = lefwMacroForeignStr("DFF3S", 0, 0, "");
CHECK_STATUS(status);
status = lefwEndMacro("DFF4");
CHECK_STATUS(status);

status = lefwStartMacro("DFF5");
CHECK_STATUS(status);
status = lefwMacroClass("COVER", NULL);
CHECK_STATUS(status);
status = lefwMacroForeignStr("DFF3S", 0, 0, "");
CHECK_STATUS(status);
status = lefwEndMacro("DFF5");
CHECK_STATUS(status);
```

```
status = lefwStartMacro("DFF6");
CHECK_STATUS(status);
status = lefwMacroClass("BLOCK", "BLACKBOX");
CHECK_STATUS(status);
status = lefwMacroForeignStr("DFF3S", 0, 0, "");
CHECK_STATUS(status);
status = lefwEndMacro("DFF6");
CHECK_STATUS(status);

status = lefwStartMacro("DFF7");
CHECK_STATUS(status);
status = lefwMacroClass("PAD", "AREAIO");
CHECK_STATUS(status);
status = lefwMacroForeignStr("DFF3S", 0, 0, "");
CHECK_STATUS(status);
status = lefwEndMacro("DFF7");
CHECK_STATUS(status);

status = lefwStartMacro("DFF8");
CHECK_STATUS(status);
status = lefwMacroClass("BLOCK", "SOFT");
CHECK_STATUS(status);
status = lefwEndMacro("DFF8");
CHECK_STATUS(status);

status = lefwStartMacro("DFF9");
CHECK_STATUS(status);
status = lefwMacroClass("CORE", "WELLTAP");
CHECK_STATUS(status);
status = lefwEndMacro("DFF9");
CHECK_STATUS(status);

status = lefwStartMacro("myTest");
CHECK_STATUS(status);
status = lefwMacroClass("CORE", NULL);
CHECK_STATUS(status);
status = lefwMacroSize(10.0, 14.0);
CHECK_STATUS(status);
status = lefwMacroSymmetry("X");
CHECK_STATUS(status);
status = lefwMacroSitePatternStr("Fsite", 0, 0, "N", 0, 0, 0, 0);
```

```
  CHECK_STATUS(status);
  status = lefwMacroSitePatternStr("Fsite", 0, 7.0, "FS", 30, 3, 2, 2);
  CHECK_STATUS(status);
  status = lefwMacroSitePatternStr("Fsite", 4.0, 0, "N", 0, 0, 0, 0);
  CHECK_STATUS(status);
  status = lefwEndMacro("myTest");
  CHECK_STATUS(status);

  // ANTENNA, this will generate error for 5.4 since I already have ANTENNA
  // somewhere
  status = lefwAntenna("INPUTPINANTENNASIZE", 1);
  CHECK_STATUS(status);
  status = lefwAntenna("OUTPUTPINANTENNASIZE", -1);
  CHECK_STATUS(status);
  status = lefwAntenna("INOUTPINANTENNASIZE", -1);
  CHECK_STATUS(status);
  status = lefwNewLine();
  CHECK_STATUS(status);

  // BEGINEXT
  status = lefwStartBeginext("SIGNATURE");
  CHECK_STATUS(status);
  lefwAddIndent();
  status = lefwBeginextCreator("CADENCE");
  CHECK_STATUS(status);
  status = lefwEndBeginext();
  CHECK_STATUS(status);

  status = lefwEnd();
  CHECK_STATUS(status);

  lineNum = lefwCurrentLineNumber();
  if (lineNum == 0)
     fprintf(stderr, "ERROR: Nothing has been written!!!\n");

  fclose(fout);

  return 0;
}
```