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

Product Version 5.8
May 2017

Contents

Preface 15

What's New 15

Related Documents 15

Typographic and Syntax Conventions 15

1

Introduction 17

Overview 17

DEF Reader Working Modes 17

Comparison Utility 18

Compressed DEF Files 19

Orientation Codes 19

2

DEF Reader Setup and Control Routines 21

DEF API Routines 21

- defrInit 22
- defrInitSession 22
- defrClear 22
- defrRead 23
- defrSetUserData 23
- defrGetUserData 24
- defrSetAddPathToNet 24
- defrSetAllowComponentNets 24
- defrGetAllowComponentNets 25
- defrSetCommentChar 25
- defrSetRegisterUnusedCallbacks 25
- defrPrintUnusedCallbacks 25
- defrUnusedCallbackCount 26
- Example 26

3
DEF Reader Callback Routines 29
Callback Function Format 29
Callback Type 29
DEF_Data 30
User Data 30
Callback Types and Setting Routines 30
Examples 35
User Callback Routines 36
defrBlockageCbkFnType 37
defrBoxCbkFnType 38
defrComponentCbkFnType 39
defrComponentMaskShiftLayerCbkFnType 39
defrDoubleCbkFnType 40
defrFillCbkFnType 41
defrGcellGridCbkFnType 42
defrGroupCbkFnType 43
defrIntegerCbkFnType 43
defrNetCbkFnType 45
defrNonDefaultCbkFnType 46
defrPathCbkFnType 47
defrPinCbkFnType 47
defrPinPropCbkFnType 48
defrPropCbkFnType 49
defrRegionCbkFnType 49
defrRowCbkFnType 50
defrScanchainCbkFnType 51
defrSlotCbkFnType 51
defrStringCbkFnType 52
defrStylesCbkFnType 54
defrTrackCbkFnType 55
defrViaCbkFnType 55
defrVoidCbkFnType 56
Examples 58

4 DEF Reader Classes

DEF Reader Classes 61
Introduction 61
Callback Style Interface 61
Retrieving Repeating DEF Data 62
Deriving C Syntax from C++ Syntax 62
  C++ Syntax 62
  C Syntax 62
DEF Reader Class Routines 63
  defiBlockage 65
  defiBox 65
  defiComponent 66
  defiComponentMaskShiftLayer 69
  defiFill 69
  defiGcellGrid 70
  defiGeometries 71
  defiGroup 71
  defiNet 73
  defiNonDefault 77
  defiOrdered 78
  defiPath 78
  defiPin 79
  defiPinAntennaModel 82
  defiPinPort 83
  defiPinProp 83
  defiPoints 85
  defiProp 85
  defiRegion 87
  defiRow 88
  defiScanchain 90
  defiShield 92
  defiSite 92
  defiSlot 93
  defiStyles 94
  defiSubnet 94

defiTrack 95
defiVia 96
defiViaData 98
defiVpin 98
defiWire 98

5
DEF Writer Callback Routines 101
Callback Function Format 102
Callback Type 102
User Data 102
Callback Types and Setting Routines 103

6
DEF Writer Routines 105
DEF Writer Setup and Control 106
defwInit 106
defwInitCbk 108
defwEnd 108
defwCurrentLineNumber 108
defwNewLine 109
defwAddComment 109
defwAddIntent 109
defwPrintError 109
Setup Examples 110
Blockages 112
defwStartBlockages 112
defwEndBlockages 113
defwBlockageDesignRuleWidth 113
defwBlockagesLayerDesignRuleWidth 114
defwBlockageLayer 114
defwBlockagesLayer 115
defwBlockagesLayerComponent 115
defwBlockageLayerExceptpgnet 116
defwBlockagesLayerExceptpgnet 116

- defwBlockageLayerFills 117
- defwBlockagesLayerFills 117
- defwBlockageLayerPushdown 117
- defwBlockagesLayerPushdown 118
- defwBlockageLayerSlots 118
- defwBlockagePlacement 119
- defwBlockagesPlacement 119
- defwBlockagePlacementComponent 119
- defwBlockagesPlacementComponent 120
- defwBlockagePlacementPartial 121
- defwBlockagesPlacementPartial 121
- defwBlockagePlacementPushdown 122
- defwBlockagesPlacementPushdown 122
- defwBlockagePlacementSoft 122
- defwBlockagesPlacementSoft 123
- defwBlockagePolygon 123
- defwBlockagesPolygon 124
- defwBlockageRect 124
- defwBlockagesRect 125
- defwBlockagesLayerMask 125
- defwBlockageSpacing 126

Bus Bit Characters 126
- defwBusBitChars 126

Components 127
- defwStartComponents 127
- defwEndComponents 128
- defwComponent 128
- defwComponentStr 130
- defwComponentHalo 133
- defwComponentHaloSoft 133
- defwComponentRouteHalo 134

Design Name 135
- defwDesignName 136

Die Area 136
- defwDieArea 136
- defwDieAreaList 137

Die Area Example 137
Divider Character 138
defwDividerChar 138
Extensions 138
defwStartBeginext 139
defwEndBeginext 139
defwBeginextCreator 139
defwBeginextDate 140
defwBeginextRevision 140
defwBeginextSyntax 140
Extensions Example 141
Fills 141
defwStartFills 142
defwEndFills 142
defwFillLayer 142
defwFillLayerOPC 143
defwFillPoints 143
defwFillPolygon 143
defwFillRect 144
defwFillVia 144
defwFillViaOPC 145
GCell Grid 145
defwGcellGrid 145
Gcell Grid Example 146
Groups 147
defwStartGroups 147
defwEndGroups 147
defwGroup 148
defwGroupRegion 148
History 150
defwHistory 150
Nets 151
defwStartNets 151
defwEndNets 152
defwNet 152
defwNetMustjoinConnection 152

- defwNetEndOneNet 153
- defwNetConnection 153
- defwNetEstCap 154
- defwNetFixedBump 154
- defwNetFrequency 154
- defwNetNondefaultRule 155
- defwNetOriginal 155
- defwNetPattern 156
- defwNetSource 156
- defwNetUse 157
- defwNetVpin 158
- defwNetVpinStr 159
- defwNetWeight 160
- defwNetXtalk 161
- Nets Example 161

Regular Wiring 164
- defwNetPathStart 164
- defwNetPathEnd 165
- defwNetPathLayer 165
- defwNetPathPoint 166
- defwNetPathStyle 167
- defwNetPathVia 167
- defwNetPathViaWithOrient 168
- defwNetPathViaWithOrientStr 168
- Regular Wiring Example 169

Subnet 169
- defwNetSubnetStart 170
- defwNetSubnetEnd 170
- defwNetSubnetPin 170
- Subnet Example 171

Nondefault Rules 172
- defwStartNonDefaultRules 172
- defwEndNonDefaultRules 173
- defwNonDefaultRule 173
- defwNonDefaultRuleLayer 173
- defwNonDefaultRuleMinCuts 174

defwNonDefaultRuleVia 175
defwNonDefaultRuleViaRule 175
Pins 176
defwStartPins 176
defwEndPins 177
defwPin 177
defwPinStr 179
defwPinAntennaModel 182
defwPinAntennaPinDiffArea 182
defwPinAntennaPinGateArea 183
defwPinAntennaPinMaxAreaCar 183
defwPinAntennaPinMaxCutCar 184
defwPinAntennaPinMaxSideAreaCar 184
defwPinAntennaPinPartialCutArea 185
defwPinAntennaPinPartialMetalArea 185
defwPinAntennaPinPartialMetalSideArea 186
defwPinGroundSensitivity 186
defwPinLayer 187
defwPinNetExpr 187
defwPinPolygon 188
defwPinPort 189
defwPinPortLayer 189
defwPinPortLocation 190
defwPinPortPolygon 191
defwPinPortVia 192
defwPinSupplySensitivity 192
defwPinVia 193
Pins Example 193
Pin Properties 194
defwStartPinProperties 194
defwEndPinProperties 195
defwPinProperty 195
Pin Properties Example 196
Property Definitions 196
defwStartPropDef 197
defwEndPropDef 197

- defwIntPropDef 197
- defwRealPropDef 198
- defwStringPropDef 199
- Property Definitions Example 199

Property Statements 200
- defwIntProperty 200
- defwRealProperty 201
- defwStringProperty 201
- Property Statements Example 202

Regions 202
- defwStartRegions 203
- defwEndRegions 203
- defwRegionName 203
- defwRegionPoints 204
- defwRegionType 204
- Regions Example 205

Rows 206
- defwRow 206
- defwRowStr 207
- Rows Example 208

Scan Chains 209
- defwStartScanchains 209
- defwEndScanchains 210
- defwScanchain 210
- defwScanchainCommonscanpins 210
- defwScanchainFloating 211
- defwScanchainFloatingBits 212
- defwScanchainOrdered 213
- defwScanchainOrderedBits 214
- defwScanchainPartition 215
- defwScanchainStart 215
- defwScanchainStop 216
- Scan Chain Example 216

Special Nets 217
- defwStartSpecialNets 218
- defwEndSpecialNets 218

- defwSpecialNet 218
- defwSpecialNetEndOneNet 219
- defwSpecialNetConnection 219
- defwSpecialNetEstCap 220
- defwSpecialNetFixedBump 220
- defwSpecialNetOriginal 220
- defwSpecialNetPattern 221
- defwSpecialNetSource 221
- defwSpecialNetUse 222
- defwSpecialNetVoltage 223
- defwSpecialNetWeight 223
- Special Nets Example 224

Special Wiring 225
- defwSpecialNetPathStart 226
- defwSpecialNetPathEnd 226
- defwSpecialNetPathLayer 227
- defwSpecialNetPathPoint 227
- defwSpecialNetPathPointWithWireExt 228
- defwSpecialNetPathShape 228
- defwSpecialNetPathStyle 229
- defwSpecialNetPathVia 229
- defwSpecialNetPathViaData 230
- defwSpecialNetPathWidth 230
- defwSpecialNetShieldNetName 231
- defwSpecialNetPolygon 231
- defwSpecialNetRect 232
- Special Wiring Example 232

Shielded Routing 233
- defwSpecialNetShieldStart 233
- defwSpecialNetShieldEnd 234
- defwSpecialNetShieldLayer 234
- defwSpecialNetShieldPoint 234
- defwSpecialNetShieldShape 235
- defwSpecialNetShieldVia 235
- defwSpecialNetShieldViaData 236
- defwSpecialNetShieldWidth 236

- Shielded Routing Example 236
- Slots 237
  - defwStartSlots 238
  - defwEndSlots 238
  - defwSlotLayer 238
  - defwSlotPolygon 239
  - defwSlotRect 239
- Styles 240
  - defwStartStyles 240
  - defwEndStyles 240
  - defwStyles 240
- Technology 241
  - defwTechnology 241
- Tracks 242
  - defwTracks 242
  - Tracks Example 243
- Units 244
  - defwUnits 244
- Version 244
  - defwVersion 244
- Vias 245
  - defwStartVias 245
  - defwEndVias 246
  - defwViaName 246
  - defwOneViaEnd 246
  - defwViaPolygon 247
  - defwViaRect 247
  - defwViaViarule 248
  - defwViaViaruleRowCol 249
  - defwViaViaruleOrigin 250
  - defwViaViaruleOffset 250
  - defwViaViarulePattern 251
  - Vias Example 251

7
DEF Compressed File Routines 253
defGZipOpen 253
defGZipClose 253
Example 254

8
DEF File Comparison Utility 255
lefdefdiff 255
Example 256

A
DEF Reader and Writer Examples 259
DEF Reader Example 259
DEF Writer Example 325

Preface

This manual describes the C and C++ programming interface used to read and write Cadence® Design Exchange Format (DEF) files. To use this manual, you should be an experienced C or C++ programmer, and be familiar with DEF file structure.

What's New

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 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 LEF and DEF for version 5.8, see *What's New in LEF/DEF*.

Related Documents

The DEF C/C++ programming interface lets you create programs that read and write DEF 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.

4/18/17

1

Introduction

This chapter contains the following sections:

- Overview
- DEF Reader Working Modes
- Comparison Utility
- Compressed DEF Files on page 19
- Orientation Codes on page 19

Overview

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

- DEF reader
- DEF writer

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

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 DEF output.

DEF Reader Working Modes

The DEF 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 `defrInit()`, 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 `defrInitSession()` instead of `defrInit()` 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 DEF files with the `defrRead()` function.

  d. Cleaning of the parsing configuration: Call the `defrClear()` 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 `defrInitSession()` call.

  In the session-based mode, the properties once defined in `PROPERTYDEFINITIONS` remain active in all the DEF 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 `defrInitSession()` 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 DEF parser works in the compatibility mode. To activate the session-based mode, you must use `defrInitSession()` instead of `defrInit()`.

Note: Currently, the compatibility mode can be used in all old applications where the code has not been adjusted. The `def2oa` translator has already been adjusted to use the session-based parsing mode.

Comparison Utility

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

Because the DEF file comparison utility works incrementally (writing out as it operates), the size of files it can process has no limitation. However, large files can have performance restrictions. In general, the 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 DEF Files

The DEF reader can parse compressed DEF files. To do so, you must link the `libdef.a` and `libdefzlib.a` libraries.

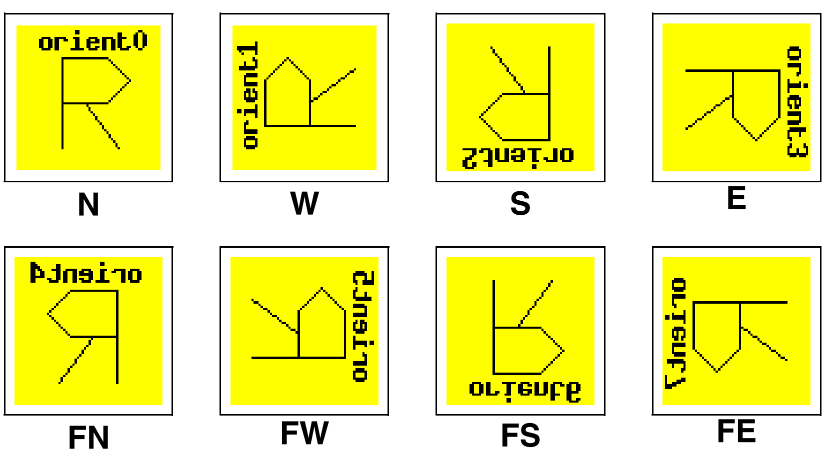
A zlib compression library is also required in order to read compressed DEF files. The zlib source code is free software that can be downloaded from www.gnu.com.

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

Orientation Codes

Orientation codes are used throughout the DEF 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 4 = FN
orient 1 = W	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

DEF Reader Setup and Control Routines

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

The following routines described in this section set options for reading a DEF file.

- defrInit on page 22
- defrInitSession on page 22
- defrClear on page 22
- defrRead on page 23
- defrSetUserData on page 23
- defrGetUserData on page 24
- defrSetAddPathToNet on page 24
- defrSetAllowComponentNets on page 24
- defrGetAllowComponentNets on page 25
- defrSetCommentChar on page 25
- defrSetRegisterUnusedCallbacks on page 25
- defrPrintUnusedCallbacks on page 25
- defrUnusedCallbackCount on page 26

DEF API Routines

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

defrInit

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

Syntax

```
int defrInit()
```

defrInitSession

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

Syntax

```
int defrInitSession (
     int startSession = 1)
```

Arguments

`startSession`

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

defrClear

Releases all parsing session data and closes the parsing session. If the call to `defrClear()` is skipped, the data cleaning and the session closing is done by the next `defrInitSession()` call.

Syntax

```
int defrClear()
```

defrRead

Specifies the DEF file to read. Any callbacks that have been set are called from within this routine. If the file parses with no errors, that is, all callbacks return OK condition codes, this routine returns zero.

Syntax

```
int defrRead(
     FILE* file,
     const char* fileName,
     defiUserData* data,
     int case_sensitive)
```

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.

case_sensitive

Specifies whether the data is case sensitive.

defrSetUserData

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

Syntax

```
void defrSetUserData(
     defiUserData* data)
```

Arguments

data

Specifies the user-provided data.

defrGetUserData

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

Syntax

```
defiUserData defrGetUserData()
```

defrSetAddPathToNet

Adds path data to the appropriate net data. When the net callback is used, the net class and structure information and the path information are returned. This statement does not require any additional arguments.

Syntax

```
void defrSetAddPathToNet(void)
```

defrSetAllowComponentNets

Ignores component net information. Component nets are valid DEF syntax but are no longer used. By default, the DEF reader reports component net data as a syntax error. This routine overrides the default so no error is reported. This statement does not require any additional arguments.

Syntax

```
void defrSetAllowComponentNets(void)
```

defrGetAllowComponentNets

Returns non-zero value if component nets are allowed.

Syntax

```
int defrGetAllowComponentNets()
```

defrSetCommentChar

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

Syntax

```
void defrSetCommentChar(char c)
```

Arguments

c

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

defrSetRegisterUnusedCallbacks

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

Syntax

```
void defrSetRegisterUnusedCallbacks(void)
```

defrPrintUnusedCallbacks

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

Syntax

```
void defrPrintUnusedCallbacks(FILE* log)
```

Arguments

log

Specifies the file to which the unused callbacks are printed.

defrUnusedCallbackCount

Returns the number of callback routines that are not set. That is, routines that have constructs in the input file but no callback trigger. This statement does not require any additional arguments.

Syntax

```
int* defrUnusedCallbackCount(void)
```

Example

The following example shows how to initialize the reader.

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

          //   Initialize the reader. This routine has to call first.
          defrInit();

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

          // Open the def file for the reader to read
          if ((f = fopen("defInputFileName","r")) == 0) {
               printf("Couldn't open input file '%s'\n",
               "defInputFileName");
               return(2);
          }
          // Invoke the parser
          res = defrRead(f, "defInputFileName", (void*)userData);
          if (res != 0) {
               printf("DEF parser returns an error\n");
               return(2);
```

```
}
fclose(f);
return 0;}
```

3

DEF Reader Callback Routines

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

This chapter contains the following sections:

- Callback Function Format
- Callback Types and Setting Routines on page 30
- User Callback Routines on page 36

Callback Function Format

All callback functions use the following format.

```
int UserCallbackFunction(
     defrCallbackType_e callBackType
     DEF_type DEF_data
     defiUserData 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 DEF data.

DEF_Data

The *DEF_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. Note that the DEF reader 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. 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 DEF file parsing begins, you must set the callback before you call `defrRead`.

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 DEF information is passed in the appropriate setting routine. Each callback routine returns a callback type.

The following table lists the DEF 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.

DEF Information	Setting Routine	Callback Types
Blockages Beginning	`void defrSetBlockageStartCbk (defrIntegerCbkFnType)`	`defrBlockageStartCbkType`
Blockages	`void defrSetBlockageCbk (defrBlockageCbkFnType)`	`defrBlockageCbkType`

DEF Information	Setting Routine	Callback Types
Blockages End	`void defrSetBlockageEndCbk (defrVoidCbkFnType)`	`defrBlockageEndCbkType`
Bus Bit Characters	`void defrSetBusBitCbk (defrStringCbkFnType)`	`defrBusBitCbkType`
Components Beginning	`void defrSetComponentStartCbk (defrIntegerCbkFnType)`	`defrComponentStartCbkType`
Components	`void defrSetComponentCbk (defrComponentCbkFnType)`	`defrComponentCbkType`
Components End	`void defrSetComponentEndCbk (defrVoidCbkFnType`	`defrComponentEndCbkType`
Components Mask Layer	`void defrComponentMaskShiftLayerCbk (defrComponentMaskShiftLayerCbkFnType)`	
		`defrComponentMaskShiftLayerCbkType`
Constraints Path	`void defrSetPathCbk (defrPathCbkFnType`	`defrPathCbkType`
Design Beginning	`void defrSetDesignCbk (defrStringCbkFnType`	`defrDesignStartCbkType`
Design End	`void defrSetDesignEndCbk (defrVoidCbkFnType)`	`defrDesignEndCbkType`
Die Area	`void defrSetDieAreaCbk (defrBoxCbkFnType)`	`defrDieAreaCbkType`
Divider Character	`void defrSetDividerCbk (defrStringCbkFnType)`	`defrDividerCbkType`
Extensions Components	`void defrSetComponentExtCbk (defrStringCbkFnType)`	`defrComponentExtCbkType`
Extensions Groups	`void defrSetGroupExtCbk (defrStringCbkFnType)`	`defrGroupExtCbkType`
Extensions Net	`void defrSetNetExtCbk (defrStringCbkFnType)`	`defrNetExtCbkType`

DEF Information	Setting Routine	Callback Types
Extensions Net Connection	`void defrSetNetConnectionExtCbk (defrStringCbkFnType)`	`defrNetConnectionExtCbkType`
Extensions Pin	`void defrSetPinExtCbk (defrStringCbkFnType)`	`defrPinExtCbkType`
Extensions Scan Chains	`void defrSetScanChainExtCbk (defrStringCbkFnType)`	`defrScanChainExtCbkType`
Extensions Vias	`void defrSetViaExtCbk (defrStringCbkFnType)`	`defrViaExtCbkType`
Fills Beginning	`void defrSetFillStartCbk (defrIntegerCbkFnType)`	`defrFillStartCbkType`
Fills	`void defrSetFillCbk (defrFillCbkFnType)`	`defrFillCbkType`
Fills End	`void defrSetFillEndCbk (defrVoidCbkFnType)`	`defrFillEndCbkType`
GCell Grid	`void defrSetGcellGridCbk (defrGcellGridCbkFnType)`	`defrGcellGridCbkType`
Groups Beginning	`void defrSetGroupsStartCbk (defrIntegerCbkFnType)`	`defrGroupsStartCbkType`
Groups Name	`void defrSetGroupNameCbk (defrStringCbkFnType)`	`defrGroupNameCbkType`
Groups Member	`void defrSetGroupMemberCbk (defrStringCbkFnType)`	`defrGroupMemberCbkType`
Groups	`void defrSetGroupCbk (defrGroupCbkFnType)`	`defrGroupCbkType`
Groups End	`void defrSetGroupsEndCbk (defrVoidCbkFnType)`	`defrGroupsEndCbkType`
History	`void defrSetHistoryCbk (defrStringCbkFnType)`	`defrHistoryCbkType`

DEF Information	Setting Routine	Callback Types
Nets Beginning	`void defrSetNetStartCbk (defrIntegerCbkFnType)`	`defrNetStartCbkType`
Nets	`void defrSetNetCbk (defrNetCbkFnType)`	`defrNetCbkType`
Nets End	`void defrSetNetEndCbk (defrVoidCbkFnType)`	`defrNetEndCbkType`
Nondefault Rules Beginning	`void defrNonDefaultStartCbk (defrIntegerCbkFnType)`	`defrNonDefaultStartCbkType`
Nondefault Rules	`void defrSetNonDefaultCbk (defrNonDefaultCbkFnType)`	`defrNonDefaultCbkType`
Nondefault Rules End	`void defrNonDefaultEndCbk (defrVoidCbkFnType)`	`defrNonDefaultEndCbkType`
Pins Beginning	`void defrSetStartPinsCbk (defrIntegerCbkFnType)`	`defrStartPinsCbkType`
Pins	`void defrSetPinCbk (defrPinCbkFnType)`	`defrPinCbkType`
Pins End	`void defrSetPinEndCbk (defrVoidCbkFnType)`	`defrPinEndCbkType`
Pin Properties Beginning	`void defrSetPinPropStartCbk (defrIntegerCbkFnType)`	`defrPinPropStartCbkType`
Pin Properties	`void defrSetPinPropCbk (defrPinPropCbkFnType)`	`defrPinPropCbkType`
Pin Properties End	`void defrSetPinPropEndCbk (defrVoidCbkFnType)`	`defrPinPropEndCbkType`
Property Definitions Beginning	`void defrSetPropDefStartCbk (defrVoidCbkFnType)`	`defrPropDefStartCbkType`
Property Definitions	`void defrSetPropCbk (defrPropCbkFnType)`	`defrPropCbkType`

DEF Information	Setting Routine	Callback Types
Property Definitions End	`void defrSetPropDefEndCbk (defrVoidCbkFnType)`	`defrPropDefEndCbkType`
Regions Beginning	`void defrSetRegionStartCbk (defrIntegerCbkFnType)`	`defrRegionStartCbkType`
Regions	`void defrSetRegionCbk (defrRegionCbkFnType)`	`defrRegionCbkType`
Regions End	`void defrSetRegionEndCbk (defrVoidCbkFnType)`	`defrRegionEndCbkType`
Rows	`void defrSetRowCbk (defrRowCbkFnType)`	`defrRowCbkType`
Scan Chains Beginning	`void defrSetScanchainsStartCbk (defrIntegerCbkFnType)`	`defrScanchainsStartCbkType`
Scan Chains	`void defrSetScanchainCbk (defrScanchainCbkFnType)`	`defrScanchainCbkType`
Scan Chains End	`void defrSetScanchainsEndCbk (defrVoidCbkFnType)`	`defrScanchainsEndCbkType`
Slots Beginning	`void defrSetSlotStartCbk (defrIntegerCbkFnType)`	`defrSlotStartCbkType`
Slots	`void defrSetSlotCbk (defrSlotCbkFnType)`	`defrSlotCbkType`
Slots End	`void defrSlotEndCbk (defrVoidCbkFnType)`	`defrSlotEndCbkType`
Special Nets Beginning	`void defrSetSNetStartCbk (defrIntegerCbkFnType)`	`defrSNetStartCbkType`
Special Nets	`void defrSetSNetCbk (defrNetCbkFnType)`	`defrSNetCbkType`
Special Nets End	`void defrSetSNetEndCbk (defrVoidCbkFnType)`	`defrSNetEndCbkType`

DEF Information	Setting Routine	Callback Types
Styles Beginning	`void defrSetStylesStartCbk (defrIntegerCbkFnType)`	`defrStylesStartCbkType`
Styles	`void defrSetStylesCbk (defrStylesCbkFnType)`	`defrStylesCbkType`
Styles End	`void defrSetStylesEndCbk (defrVoidCbkFnType)`	`defrStylesEndCbkType`
Technology	`void defrSetTechnologyCbk (defrStringCbkFnType)`	`defrTechNameCbkType`
Tracks	`void defrSetTrackCbk (defrTrackCbkFnType)`	`defrTrackCbkType`
Units	`void defrSetUnitsCbk (defrDoubleCbkFnType)`	`defrUnitsCbkType`
Version	`void defrSetVersionCbk (defrDoubleCbkFnType)`	`defrVersionCbkType`
Version String	`void defrSetVersionStrCbk (defrStringCbkFnType)`	`defrVersionStrCbkType`
Vias Beginning	`void defrSetViaStartCbk (defrIntegerCbkFnType)`	`defrViaStartCbkType`
Vias	`void defrSetViaCbk (defrViaCbkFnType)`	`defrViaCbkType`
Vias End	`void defrSetViaEndCbk (defrVoidCbkFnType)`	`defrViaEndCbkType`

Examples

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

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

```
//   Initialize the reader. This routine has to call first.
defrInit();

// Set the user callback routines
defrSetDesignCbk(designCB);
defrSetTechnologyCbk(technologyCB);
defrSetDesignEndCbk(designEndCB);
defrSetPropCbk(propertyDefCB);
defrSetPropDefEndCbk(properyDefEndCB);
defrSetNetCbk(netCB);
...

defrSetRegisterUnusedCallback();
// Open the def file for the reader to read
if ((f = fopen("defInputFileName","r")) == 0) {
     printf("Couldn't open input file '%s'\n",
     "defInputFileName");
     return(2);
}
// Invoke the parser
res = defrRead(f, "defInputFileName", (void*)userData);
if (res != 0) {
     printf("DEF parser returns an error\n");
     return(2);
}
(void)defrPrintUnusedCallbacks(f);
fclose(f);
return 0;}
```

User Callback Routines

This section describes the following routines:

- defrBlockageCbkFnType on page 37
- defrBoxCbkFnType on page 38
- defrComponentCbkFnType on page 39
- defrComponentMaskShiftLayerCbkFnType on page 39
- defrDoubleCbkFnType on page 40
- defrFillCbkFnType on page 41
- defrGcellGridCbkFnType on page 42

- defrGroupCbkFnType on page 43
- defrIntegerCbkFnType on page 43
- defrNetCbkFnType on page 45
- defrNonDefaultCbkFnType on page 46
- defrPathCbkFnType on page 47
- defrPinCbkFnType on page 47
- defrPinPropCbkFnType on page 48
- defrPropCbkFnType on page 49
- defrRegionCbkFnType on page 49
- defrRowCbkFnType on page 50
- defrScanchainCbkFnType on page 51
- defrSlotCbkFnType on page 51
- defrStringCbkFnType on page 52
- defrStylesCbkFnType on page 54
- defrTrackCbkFnType on page 55
- defrViaCbkFnType on page 55
- defrVoidCbkFnType on page 56

defrBlockageCbkFnType

Retrieves data from the `BLOCKAGES` statement in the DEF file. Use the arguments defined in the `defiBlockage` class to retrieve the data. For syntax information about the DEF `BLOCKAGES` statement, see Blockages in the *LEF/DEF Language Reference*.

Syntax

```
int defrBlockageCbkFnType(
     defrCallbackType_e typ,
     defiBlockage* blockage,
     defiUserData* data)
```

Arguments

typ

Returns the `defrBlockageCbkType` type, which indicates that the blockage callback was called.

blockage

Returns a pointer to a `defiBlockage` structure. For more information, see defiBlockage on page 65.

data

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

defrBoxCbkFnType

Retrieves data from the `DIEAREA` statement in the DEF file. Use the arguments defined in the `defiBox` class to retrieve the data. For syntax information about the DEF `DIEAREA` statement, see Die Area in the *LEF/DEF Language Reference*.

Syntax

```
int defrBoxCbkFnType(
     defrCallbackType_e typ,
     defiBox* box,
     defiUserData* data)
```

Arguments

typ

Returns the `defrDieAreaCbkType` type, which indicates that the die area callback was called.

box

Returns a pointer to a `defiBox` structure. For more information, see defiBox on page 65.

data

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

defrComponentCbkFnType

Retrieves data from the `COMPONENTS` statement in the DEF file. Use the arguments defined in the `defiComponent` class to retrieve the data. For syntax information about the DEF `COMPONENTS` statement, see Components in the *LEF/DEF Language Reference*.

Syntax

```
int defrComponentCbkFnType(
     defrCallbackType_e typ,
     defiComponent* comp,
     defiUserData* data)
```

Arguments

typ

Returns the `defrComponentCbkType`, which indicates that the component callback was called.

comp

Returns a pointer to a `defiComponent` structure. For more information, see defiComponent on page 66.

data

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

defrComponentMaskShiftLayerCbkFnType

Retrieves data from the `COMPONENTMASKSHIFT` statement of the DEF file. 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 DEF `COMPONENTMASKSHIFT` statement, see "Component Mask Shift" in the *LEF/DEF Language Reference*.

Syntax

```
int defrComponentMaskShiftLayerCbkFnType (
     defrCallbackType_e  type,
     defiComponentMaskShiftLayer* shiftLayers,
     defiUserData* data)
```

Arguments

type

Returns the `defrComponentMaskShiftLayerCbkFnType`. This allows you to verity within your program that this is a correct callback.

shiftLayers

Returns a pointer to a `defiComponentMaskShiftLayer`. For more information, see defiComponentMaskShiftLayer on page 69.

data

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

defrDoubleCbkFnType

Retrieves data from the `UNITS` and `VERSION` statements of the DEF file. 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 DEF `UNITS` and `VERSION` statements, see Units and Version in the *LEF/DEF Language Reference*.

Note: DEF version 5.1 and later always has a version number. Earlier versions of DEF will not have a version number.

Syntax

```
int defrDoubleCbkFnType(
     defrCallbackType_e typ,
     double* number,
     defiUserData* data)
```

Arguments

typ

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

DEF Data	Type Returned
Units	`defrUnitsCbkType`

DEF Data	Type Returned
Version	`defrVersionCbkType`

`number`

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

DEF Data	Returns the Value of
Units	*DEFconvertFactor* in the `UNITS` statement
Version	*versionNumber* in the `VERSION` statement

`data`

Specifies 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 with the type `defrVersionCbkType`.

```
int versionCB (defrCallbackType_e type,
                    double versionNum,
                    defiUserData userData) {
         // Check if the type is correct
         if (type != defrVersionCbkType) {
             printf("Type is not defrVersionCbkType, terminate
             parsing.\n");
             return 1;
         }

         // Write out the version number
         printf("VERSION %g\n", versionNum);
         return 0;}
```

defrFillCbkFnType

Retrieves data from the `FILLS` statement in the DEF file. Use the arguments defined in the `defiFill` class to retrieve the data. For syntax information about the DEF `FILLS` statement, see Fills in the *LEF/DEF Language Reference*.

Syntax

```
int defrFillCbkFnType(
     defrCallbackType_e typ,
     defiFill* fill,
     defiUserData* data)
```

Arguments

typ

Returns the `defrFillCbkFnType`, which indicates that the fill callback was called.

fill

Returns a pointer to a `defifill` structure. For more information, see defiFill on page 69.

data

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

defrGcellGridCbkFnType

Retrieves data from the `GCELLGRID` statement in the DEF file. Use the arguments defined in the `defiGcellGrid` class to retrieve the data. For syntax information about the DEF `GCELLGRID` statement, see GCell Grid in the *LEF/DEF Language Reference*.

Syntax

```
int defrGcellGridCbkFnType(
     defrCallbackType_e typ,
     defiGcellGrid* grid,
     defiUserData* data)
```

Arguments

typ

Returns the `defrGcellGridCbkType`, which indicates that the gcell grid callback was called.

grid

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

data

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

defrGroupCbkFnType

Retrieves data from the `GROUPS` statement in the DEF file. Use the arguments defined in the `defiGroup` class to retrieve the data. For syntax information about the DEF `GROUPS` statement, see Groups in the *LEF/DEF Language Reference*.

Syntax

```
int defrGroupCbkFnType(
     defrCallbackType_e typ,
     defiGroup* group,
     defiUserData* data)
```

Arguments

typ

Returns the `defrGroupCbkType`, which indicates that the group callback was called.

group

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

data

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

defrIntegerCbkFnType

Marks the beginning of sections of DEF statements. The format of the data returned is always the same, but the actual data represented varies depending on the calling routine.

Syntax

```
int defrIntegerCbkFnType(
     defrCallbackType_e typ,
     int number,
     defiUserData* data)
```

Arguments

typ

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

DEF Data	Type Returned
Blockages	`defrBlockageStartCbkType`
Components	`defrComponentStartCbkType`
Fills	`defrFillStartCbkType`
Groups	`defrGroupsStartCbkType`
Nets	`defrNetStartCbkType`
Nondefault Rules	`defrNonDefaultStartCbkType`
Pin Properties	`defrPinPropStartCbkType`
Pins	`defrStartPinsCbkType`
Regions	`defrRegionStartCbkType`
Scan Chains	`defrScanchainsStartCbkType`
Slots	`defrSlotStartCbkType`
Special Nets	`defrSNetStartCbkType`
Styles	`defrStylesStartCbkType`
Vias	`defrViaStartCbkType`

number

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

DEF Data	Returns the Value of
Blockages	*numBlockages* in the `BLOCKAGES` statement
Components	*numComps* in the `COMPONENTS` statement
Fills	*numFills* in the `FILLS` statement
Groups	*numGroups* in the `GROUPS` statement
Nets	*numNets* in the `NETS` statement

DEF Data	Returns the Value of
Nondefault rules	*numRules* in the NONDEFAULTRULES statement
Pin Properties	*num* in the PINPROPERTIES statement
Pins	*numPins* in the PINS statement
Regions	*numRegions* in the REGIONS statement
Scan Chains	*numScanChains* in the SCANCHAINS statement
Slots	*numSlots* in the SLOTS statement
Special Nets	*numNets* in the SPECIALNETS statement
Styles	*numStyles* in the STYLES statement
Vias	*numVias* in the VIAS statement

data

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

defrNetCbkFnType

Retrieves data from the NETS and SPECIALNETS sections of the DEF file. Use the arguments defined in the defiNet class to retrieve the data.

For syntax information about the DEF NETS and SPECIALNETS statements, see Nets and Special Nets in the *LEF/DEF Language Reference*.

Syntax

```
int defrNetCbkFnType(
     defrCallbackType_e typ,
     defiNet* net,
     defiUserData* data)
```

Arguments

typ

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

DEF Data	Type Returned
Net	`defrNetCbkType`
Special Nets	`defrSNetCbkType`

net

Returns a pointer to a `defiNet` structure. For more information, see defiNet on page 73.

data

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

defrNonDefaultCbkFnType

Retrieves data from the `NONDEFAULTRULES` statement in the DEF file. Use the arguments defined in the `defiNonDefault` class to retrieve the data. For syntax information about the DEF `NONDEFAULTRULES` statement, see "Nondefault Rules," in the *LEF/DEF Language Reference*.

Syntax

```
int defrNonDefaultCbkFnType(
     defrCallbackType_e typ,
     defiNonDefault* rule,
     defiUserData* data)
```

Arguments

typ

Returns the `defrNonDefaultCbkType` type, which indicates that the nondefault rule callback was called.

rule

Returns a pointer to a `defiNonDefault` structure. For more information, see defiNonDefault on page 77.

data

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

defrPathCbkFnType

Retrieves data from the *regularWiring* and *specialWiring* specifications in the `NETS` and `SPECIALNETS` statements of the DEF file. Use the arguments defined in the `defiPath` class to retrieve the data.

For syntax information about the DEF `NETS` and `SPECIALNETS` statements, see Nets and Special Nets in the *LEF/DEF Language Reference*.

Syntax

```
int defrPathCbkFnType(
     defrCallbackType_e typ,
     defiPath* path,
     defiUserData* data)
```

Arguments

typ

Returns the `defrPathCbkType` type, which indicates that the path callback was called.

path

Returns a pointer to a `defiPath` structure. For more information, see defiPath on page 78.

data

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

defrPinCbkFnType

Retrieves data from the `PINS` statement in the DEF file. Use the arguments defined in the `defiPin` class to retrieve the data. For syntax information about the DEF `PINS` statement, see Pins in the *LEF/DEF Language Reference*.

Syntax

```
int defrPinCbkFnType(
     defrCallbackType_e typ,
     defiPin* pin,
     defiUserData* data)
```

Arguments

typ

Returns the `defrPinCbkType` type, which indicates that the Pin callback was called.

pin

Returns a pointer to a `defiPin` structure. For more information, see defiPin on page 79.

data

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

defrPinPropCbkFnType

Retrieves data from the `PINPROPERTIES` statement in the DEF file. Use the arguments defined in the `defiPinProp` class to retrieve the data. For syntax information about the DEF `PINPROPERTIES` statement, see Pin Properties in the *LEF/DEF Language Reference*.

Syntax

```
int defrPinPropCbkFnType(
     defrCallbackType_e typ,
     defiPinProp* pp,
     defiUserData* data)
```

Arguments

typ

Returns the `defrPinPropCbkType` type, which indicates that the pin property callback was called.

pp

Returns a pointer to a `defiPinProp` structure. For more information, see defiPinProp on page 83.

data

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

defrPropCbkFnType

Retrieves data from the `PROPERTYDEFINITIONS` statement in the DEF file. Use the arguments defined in the `defiProp` class to retrieve the data. For syntax information about the DEF `PROPERTYDEFINITIONS` statement, see Property Definitions in the *LEF/DEF Language Reference*.

Syntax

```
int defrPropCbkFnType(
     defrCallbackType_e typ,
     defiProp* prop,
     defiUserData* data)
```

Arguments

typ

Returns the `defrPropCbkType` type, which indicates that the property callback was called.

prop

Returns a pointer to a `defiProp` structure. For more information, see defiProp on page 85.

data

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

defrRegionCbkFnType

Retrieves data from the `REGIONS` statement in the DEF file. Use the arguments defined in the `defiRegion` class to retrieve the data. For syntax information about the DEF `REGIONS` statement, see Regions in the *LEF/DEF Language Reference*.

Syntax

```
int defrRegionCbkFnType(
     defrCallbackType_e typ,
     defiRegion* reg,
     defiUserData* data)
```

Arguments

typ

Returns the `defrRegionCbkType` type, which indicates that the region callback was called.

reg

Returns a pointer to a `defiRegion` structure. For more information, see defiRegion on page 87.

data

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

defrRowCbkFnType

Retrieves data from the `ROWS` statement in the DEF file. Use the arguments defined in the `defiRow` class to retrieve the data. For syntax information about the DEF `ROWS` statement, see Rows in the *LEF/DEF Language Reference*.

Syntax

```
int defrRowCbkFnType(
     defrCallbackType_e typ,
     defiRow* row,
     defiUserData* data)
```

Arguments

typ

Returns the `defrRowCbkType` type, which indicates that the row callback was called.

row

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

data

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

defrScanchainCbkFnType

Retrieves data from the `SCANCHAINS` statement in the DEF file. Use the arguments defined in the `defiScanchain` class to retrieve the data. For syntax information about the DEF `SCANCHAINS` statement, see Scan Chains in the *LEF/DEF Language Reference*.

Syntax

```
int defrScanchainCbkFnType(
     defrCallbackType_e typ,
     defiScanchain* sc,
     defiUserData* data)
```

Arguments

typ

Returns the `defrScanchainCbkType` type, which indicates that the scan chains callback was called.

sc

Returns a pointer to a `defiScanchain` structure. For more information, see defiScanchain on page 90.

data

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

defrSlotCbkFnType

Retrieves data from the `SLOTS` statement in the DEF file. Use the arguments defined in the `defiSlot` class to retrieve the data. For syntax information about the DEF `SLOTS` statement, see Slots in the *LEF/DEF Language Reference*.

Syntax

```
int defrSlotCbkFnType(
     defrCallbackType_e typ,
     defiSlot* slot,
     defiUserData* data)
```

Arguments

typ

Returns the type, `defrSlotCbkFnType`, which indicates that the slot callback was called.

slot

Returns a pointer to a `defiSlot` structure. For more information, see defiSlot on page 93.

data

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

defrStringCbkFnType

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.

Syntax

```
int defrStringCbkFnType(
     defrCallbackType_e typ,
     const char* string,
     defiUserData* data)
```

Arguments

typ

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

DEF Data	Type Returned
Bus Bit Characters	`defrBusBitCbkType`

DEF Data	Type Returned
Design	`defrDesignStartCbkType`
Component Extension	`defrComponentExtCbkType`
Divider Character	`defrDividerCbkType`
Group Extension	`defrGroupExtCbkType`
Groups Member	`defrGroupMemberCbkType`
Groups Name	`defrGroupNameCbkType`
History	`defrHistoryCbkType`
Net Connection Extension	`defrNetConnectionExtCbkType`
Net Extension	`defrNetExtCbkType`
Pin Extension	`defrPinExtCbkType`
Scan Chain Extension	`defrScanChainExtCbkType`
Technology	`defrTechNameCbkType`
Version	`defrVersionStrCbkType`
Via Extension	`defrViaExtCbkType`

`string`

The data returned varies depending on the callback used.The following table shows the kinds of data returned.

DEF Data	Returns a Value of
Bus Bit Characters	*delimiterPair* in the `BUSBITCHARS` statement
Design	*designName* in the `DESIGN` statement
Component Extension	*tag* in the `EXTENSIONS` statement
Divider Character	*character* in the `DIVIDERCHAR` statement
Group Extension	*tag* in the `EXTENSION` statement
Groups Member	*`compNameRegExpr`* in the `GROUPS` statement
Groups Name	*`groupName`* in the `GROUPS` statement
History	*anyText* in the `HISTORY` statement

DEF Data	Returns a Value of
Net Connection Extension	*tag* in the `EXTENSION` statement
Net Extension	*tag* in the `EXTENSION` statement
Pin Extension	*tag* in the `EXTENSION` statement
Scan Chain Extension	*tag* in the `EXTENSION` statement
Technology	*technologyName* in the `TECHNOLOGY` statement
Version	*`versionNumber`* in `VERSION` statement
Via Extension	*tag* in the `EXTENSION` statement

`data`

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

defrStylesCbkFnType

Retrieves data from the `STYLES` statement in the DEF file. Use the arguments defined in the `defiStyles` class to retrieve the data. For syntax information about the DEF `STYLES` statement, see "Styles," in the *LEF/DEF Language Reference*.

Syntax

```
defrStylesCbkFnType(
     defCallbackType_e typ,
     defiStyles* style,
     defiUserData* data)
```

Arguments

`typ`

Returns the `defrStylesCbkType`, which indicates that the style callback was called.

`style`

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

data

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

defrTrackCbkFnType

Retrieves data from the `TRACKS` statement in the DEF file. Use the arguments defined in the `defiTrack` class to retrieve the data. For syntax information about the DEF `TRACKS` statement, see Tracks in the *LEF/DEF Language Reference*.

Syntax

```
int defrTrackCbkFnType(
     defrCallbackType_e typ,
     defiTrack* track,
     defiUserData* data)
```

Arguments

typ

Returns the `defrTrackCbkType`, which indicates that the track callback was called.

sc

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

data

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

defrViaCbkFnType

Retrieves data from the `VIAS` statement in the DEF file. Use the arguments defined in the `defiVia` class to retrieve the data. For syntax information about the DEF `VIAS` statement, see Vias in the *LEF/DEF Language Reference*.

Syntax

```
int defrViaCbkFnType(
     defrCallbackType_e typ,
     defiVia* via,
     defiUserData* data)
```

Arguments

typ

Returns the `defrViaCbkType`, which indicates that the via callback was called.

via

Returns a pointer to a `defiVia` structure. For more information, see defiVia on page 96.

data

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

defrVoidCbkFnType

Marks the end of DEF data sections. The format of the data returned is always the same, but the actual data represented varies depending on the calling routine.

Syntax

```
int defrVoidCbkFnType(
     defrCallbackType_e typ,
     void* variable,
     defiUserData* data)
```

Arguments

typ

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

DEF Data	Type Returned
Blockages, End	`defrBlockageEndCbkType`
Component, End	`defrComponentEndCbkType`

DEF Data	Type Returned
Design, End	`defrDesignEndCbkType`
Fills, End	`defrFillEndCbkType`
Groups, End	`defrGroupsEndCbkType`
Net, End	`defrSNetEndCbkType`
Nondefault Rules, End	`defrNonDefaultEndCbkType`
Pin Properties, End	`defrPinPropEndCbkType`
Pins, End	`defrPinEndCbkType`
Property Definitions, End	`defrPropDefEndCbkType`
Property Definitions, Start	`defrPropDefStartCbkType`
Region, End	`defrRegionEndCbkType`
Scan Chains, End	`defrConstraintsEndCbkType`
Slots, End	`defrSlotEndCbkType`
Special Nets, End	`defrSNetEndCbkType`
Styles, End	`defrStylesEndCbkType`
Via, End	`defrViaEndCbkType`

`variable`

Returns data that varies depending on the callback used. The following kinds of data can be returned. For all data types, the variable returns `NULL`.

DEF Data
Blockages, End
Component, End
Design, End
Fills, End
Groups, End
Net, End
Nondefault Rules, End

DEF Data
Pins, End
Pin Properties, End
Property Definitions, End
Property Definitions Start
Region, End
Scan Chains, End
Slots, End
Special Nets, End
Styles, End
Via, End

`data`

Specifies 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 the arguments for `defrCallbackType_e`, `char*`, and `defiUserData`.

```
int designCB (defrCallbackType_e type,
     const char *designName,
     defiUserData userData) {

     // Incorrect type was passed in, expecting the type defrDesignStartCbk

     Type
         if (type != defrDesignStartCbkType) {
             printf("Type is not defrDesignStartCbkType,
             terminate parsing.\n");

               return 1;}

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

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

The following example shows a callback routine using the arguments for `defrCallbackType_e`, `int`, and `defiUserData`.

```
int viaStartCB (defrCallbackType_e c,
     int numVias,
     defiUserData ud) {


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

     printf("VIA %d\n", numVias);

     return 0;}
```

The following example shows a callback routine using the arguments for `defrCallbackType_e`, `defiVia`, and `defiUserData`.

```
int viaCB (defrCallbackType_e type,
               defiVia *viaInfo,
               defiUserData userData) {
         int  i, xl, yl, xh, yh;
         char *name

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


     printf("VIA %s\n", viaInfo->name());
         if (viaInfo->hasPattern())
             printf("  PATTERNNAME %s\n", viaInfo->pattern());
         for (i = 0; i < viaInfo->numLayers(); i++) {
             viaInfo->layer(i, &name, &xl, &yl, &xh, &yh);
              printf(" RECT %s %d %d %d %d\n", name, xl, yl, xh, yh);}


     return 0;}
```

4

DEF Reader Classes

This chapter contains the following sections:

- Introduction
- Callback Style Interface
- Retrieving Repeating DEF Data on page 62
- Deriving C Syntax from C++ Syntax on page 62
- DEF Reader Class Routines on page 63

Introduction

Every statement in the Cadence® Design Exchange Format (DEF) file is associated with a DEF reader class. When the DEF 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 DEF file.

Callback Style Interface

This programming interface uses a callback style interface. You register for the constructs that interest you, and the reader calls 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 DEF Data

Many DEF objects contain repeating objects or specifications. The classes that correspond to these DEF 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< A->defiVia::numLayers(); i++) {
     via -> defiVia::layer(i, &name, &x1, &y1, &xh, &yh);
     printf("+ RECT %s %d %d %d %d \n", name x1, y1, xh, yh);
```

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 objects in C and C++ syntax.

C++ Syntax

```
class defiVia {
     const char* name() const;
     const char* pattern() const;
     int hasPattern() const;
     int numLayers() const;

     void layer(int index, char** layer, int* xl, int* yl,
          int* xh, int* yh) const;}
```

C Syntax

```
const char * defiVia)name
     ( const defiVia * this );

const char * defiVia_hasPattern
     ( const defiVia * this );

int defiVia_hasPattern
     ( const defiVia * this );
```

```
int defiVia_numLayers
     ( const defiVia * this );

void defiVia_layer
     ( const defiVia * this,
          int index,
          char **layer,
          int *x1
          int *y1
          int *xh
          int *yh);
```

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

defiArray.h	defiNonDefault.h	defiViaRule.h
defiCrossTalk.h	defrCallBacks.h	defiProp.h
defrReader.h	defiDebug.h	defiDefs.h
defwWriter.h	defiLayer.h	defiUnits.h
defiUser.h	defiMacro.h	defiUtil.h
defiMisc.h	defiVia.h	

DEF Reader Class Routines

The following table lists the class routines that apply to the DEF information.

DEF Information	DEF Class
Blockages	defiBlockage
Components	defiComponent defiProp defiComponentMaskShiftLayer
Fills	defiFill
GCell Grid	defiGcellGrid
Groups	defiGroup defiProp

DEF Information	DEF Class
Nets	defiNet defiPath defiProp defiSubnet defiVpin defiWire
Nondefault Rules	defiNonDefault
Pins	defiPin defiPinAntennaModel defiProp
Pin Properties	defiPinProp
Regions	defiRegion defiProp
Rows	defiProp defiRow defiSite
Scan Chains	defiOrdered defiScanchain
Slots	defiSlot
Special Nets	defiNet defiPath defiProp defiShield defiViaData defiWire
Styles	defiStyles
Tracks	defiTrack
Vias	defiVia
Miscellaneous	defiBox defiGeometries defiPoints `defiUser` (defined as void; can be any user-defined pointer)

defiBlockage

Retrieves data from the BLOCKAGES statement in the DEF file. For syntax information about the DEF BLOCKAGES statement, see "Blockages" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiBlockage {
     int hasLayer() const;
     int hasPlacement() const;
     int hasComponent() const;
     int hasSlots() const;
     int hasFills() const;
     int hasPushdown() const;
     int hasExceptpgnet() const;
     int hasSoft() const;
     int hasPartial() const;
     int hasSpacing() const;
     int hasDesignRuleWidth() const;
     int minSpacing() const;
     int designRuleWidth() const;
     double placementMaxDensity() const;
     const char* layerName() const;
     const char* layerComponentName() const;
     const char* placementComponentName() const;

     int numRectangles() const;
     int xl(int index) const;
     int yl(int index) const;
     int xh(int index) const;
     int yh(int index) const;

     int numPolygons() const;
     struct defiPoints getPolygon(int index) const;
     int hasMask() const;
     int mask() const;}
```

defiBox

Retrieves data from the DIEAREA statement of the DEF file. For syntax information about the DEF DIEAREA statement, see "Die Area" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiBox {
     int xl() const;
     int yl() const;
     int xh() const;
     int yh() const;

     struct defiPoints getPoint() const;}
```

defiComponent

Retrieves data from the `COMPONENTS` statement in the DEF file. For syntax information about the DEF `COMPONENTS` statement, see "Components" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiComponent {
     const char* id() const;
     const char* name() const;
     int placementStatus() const;
     int isUnplaced() const;
     int isPlaced() const;
     int isFixed() const;
     int isCover() const;
     int placementX() const;
     int placementY() const;
     int placementOrient() const;          // optional- For information,see
                                           //"Orientation Codes" on page 19
     const char* placementOrientStr() const;
     int hasRegionName() const;
     int hasRegionBounds() const;
     int hasEEQ() const;
     int hasGenerate() const;
     int hasSource() const;
     int hasWeight() const;
     int weight() const;
     int hasNets() const;
     int numNets() const;
     const char* net(int index) const;
     const char* regionName() const;
     const char* source() const;
     const char* EEQ() const;
     const char* generateName() const;
     const char* macroName() const;
     int hasHalo() const;
     int hasHaloSoft() const;
```

```
int hasRouteHalo() const;
int haloDist() const;
const char* minLayer() const;
const char* maxLayer() const;
void haloEdges(int* left, int* bottom, int* right, int* top);

void regionBounds(int* size, int** xl, int** yl, int** xh, int** yh);

int hasForeignName() const;
const char* foreignName() const;
int foreignX() const;
int foreignY() const;
const char* foreignOri() const;
int hasFori() const;
int foreignOrient() const;

int numProps() 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;
int maskShiftSize();
int maskShift(int index) const;}
```

Examples

The following example shows a callback routine with the type `defrComponentCbkType`. Callback routines for the type `defrComponentStartCbkType` and `defrComponentEndCbkType` are similar to the example for `defrViaStartCbkType` and `defrViaEndCbkType` in the Via section.

```
int componentCB (defrCallbackType_e type,
                      defiComponent* compInfo,
                      defiUserData userData) {

     int i;

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

     printf("%s %s ", compInfo->id(), compInfo->name());
         if (compInfo->hasNets()) {
             for (i = 0; i < compInfo->numNets(); i++)
```

```
            printf("%s ", compInfo->net(i));
        printf("\n");
    }
    if (compInfo->isFixed())
        printf(" FIXED %d %d %d\n", compInfo->placementX(),
               compInfo->placementY(),
               compInfo->placementOrient());
    if (compInfo->isCover())
        printf("  COVER %d %d %d\n", compInfo->placementX(),
               compInfo->placementY(),
               compInfo->placementOrient());
    if (compInfo->isPlaced())
        printf(fout,"  PLACED %d %d %d\n", compInfo->placementX(),
               compInfo->placementY(),
               compInfo->placementOrient());
    if (compInfo->hasSource())
        printf("  SOURCE %s\n", compInfo->source());
    if (compInfo->hasWeight())
        printf("  WEIGHT %d\n", compInfo->weight());
    if (compInfo->hasEEQ())
        printf("  EEQMASTER %s\n", compInfo->EEQ());
    if (compInfo->hasRegionName())
        printf("  REGION %s\n", compInfo->regionName());
    if (compInfo->hasRegionBounds()) {
        int *xl, *yl, *xh, *yh;
        int size;
        compInfo->regionBounds(&size, &xl, &yl, &xh, &yh);
        for (i = 0; i < size; i++) {
            printf("  REGION %d %d %d %d\n", xl[i], yl[i],
                   xh[i], yh[i]);
        }
    }
    if (compInfo->hasForeignName()) {
        printf("  FOREIGN %s %d %d %s\n", compInfo->foreignName(),
               compInfo->foreignX(), compInfo->foreignY(),
               compInfo->foreignOri());
    }
    // maskShiftArray[0] will always return the right most digit,since we
    // allow the leading 0 and also omit the leading 0's.
    if (compInfo->maskShiftSize()) {
       printf(" MASKSHIFT");

       for (i = compInfo->maskShiftSize() -1; i >=0; i--) {
           printf("%d ", compInfo->maskShift(i);
       }
       printf("\n");
    }
return 0;
}
```

defiComponentMaskShiftLayer

Retrieves data from the `COMPONENTMASKSHIFT` statement in the DEF file.

For syntax information about the DEF `COMPONENTMASKSHIFT` statement, see "Component Mask Shift" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiComponentMaskShiftLayer {
     public:
          defiComponentMaskShiftLayer();
          ~defiComponentMaskShiftLayer();
     void Init();
     void Destroy();
     void addMaskShiftLayer(const char* layer);
     int numMaskShiftLayers() const;
     void bumpLayers(int size);
     void clear();
     const char* maskShiftLayer(int index) const;};
```

defiFill

Retrieves data from the `FILLS` statement in the DEF file. For syntax information about the DEF `FILLS` statement, see "Fills" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiFill {
     int hasLayer() const;
     const char* layerName() const;
     int hasLayerOpc() const;
     int numRectangles() const;
     int xl(int index) const;
     int yl(int index) const;
     int xh(int index) const;
     int yh(int index) const;
     int numPolygons() const;
     struct defiPoints getPolygon(int index) const;
     int hasVia() const;
     const char* viaName() const;
     int hasViaOpc() const;

     int numViaPts() const;
     struct defiPoints getViaPts(int index) const;
```

```
     void setMask(int colorMask);
     int layerMask() const
     int viaTopMask() const;
     int viaCutMask() const;
     int viaBottomMask() const;}
```

defiGcellGrid

Retrieves data from the `GCELLGRID` statement in the DEF file. For syntax information about the DEF `GCELLGRID` statement, see "GCell Grid" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiGcellGrid {
     const char* macro() const;
     int x() const;
     int xNum() const;
     double xStep() const;}
```

Examples

The following example shows a callback routine with the type `defrGcellGridCbkType`, and the class `defiGcellGrid`.

```
int gcellCB (defrCallbackType_e type,
                  defiGcellGrid* gcellInfo,
                  defiUserData userData) {
         int i;

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

         printf("GCELLGRID %s %d DO %d STEP %g\n", gcellInfo->macro(),
                gcellInfo->x(), gcellInfo->xNum(), gcellInfo->xStep());
         return 0;
     }
```

defiGeometries

Retrieves geometry data from the `BLOCKAGES`, `FILLS`, `NETS`, and `SLOTS` statements of the DEF file. For syntax information, see "Blockages," "Fills," "Nets," and "Slots" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiGeometries {
     int numPoints() const;
     void points(int index, int* x, int* y);}
```

defiGroup

Retrieves data from the `GROUPS` statement in the DEF file. For syntax information about the DEF `GROUPS` statement, see "Groups" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiGroup {
     const char* name() const;
     const char* regionName() const;
     int hasRegionBox() const;
     int hasRegionName() const;
     int hasMaxX() const;
     int hasMaxY() const;
     int hasPerim() const;
     void regionRects(int* size, int** xl, int** yl, int** xh, int** yh);
     int maxX() const;
     int maxY() const;
     int perim() 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; }
```

Examples

The following example shows callback routines for the types `defrGroupNameCbkType`, `defrGroupMemberCbkType`, and `defrGroupCbkType`. Callback routines for the type `defrGroupsStartCbkType` and `defrGroupsEndCbkType` are similar to the example for `defrViaStartCbkType` and `defrViaEndCbkType` in the Via section.

```
int groupnameCB (defrCallbackType_e type,
                      const char* name,
                      defiUserData userData) {

     // Check if the type is correct
         if ((type != defrGroupNameCbkType)) {
             printf("Type is not defrGroupNameCbkType terminate
             parsing.\n");
             return 1;
         }
         printf("Name is %s\n", name());
         return 0;
     }

int groupmemberCB (defrCallbackType_e type,
                        const char* name,
                        defiUserData userData) {
         // Check if the type is correct
         if ((type != defrGroupMemberCbkType)) {
             printf("Type is not defrGroupMemberCbkType terminate
             parsing.\n");
             return 1;
         }
         printf("  %s\n", name());
         return 0;
     }


int groupCB (defrCallbackType_e type,
                  defiGroup grouInfo,
                  defiUserData userData) {
         // Check if the type is correct
         if ((type != defrGroupCbkType)) {
             printf("Type is not defrGroupCbkType terminate
             parsing.\n");
             return 1;
         }
         if (group->hasMaxX() | group->hasMaxY() |
             group->hasPerim())
         {
             printf("  SOFT ");
             if (group->hasPerim())
                 printf("MAXHALFPERIMETER %d ", group->perim());
             if (group->hasMaxX())
                 printf("MAXX %d ", group->maxX());
             if (group->hasMaxY())
                 printf("MAXY %d ", group->maxY());
         }
         if (group->hasRegionName())
```

```
        printf("REGION %s ", group->regionName());
    if (group->hasRegionBox()) {
        int *gxl, *gyl, *gxh, *gyh;
        int size;
        group->regionRects(&size, &gxl, &gyl, &gxh, &gyh);
        for (i = 0; i < size; i++)
            printf("REGION %d %d %d %d ", gxl[i], gyl[i], gxh[i],
           gyh[i]);
    }
    printf("\n");
    return 0;}
```

defiNet

Retrieves data from the `NETS` statement in the DEF file. For syntax information about the DEF `NETS` statement, see "Nets" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiNet {
     const char* name() const;
     int weight() 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 numConnections() const;
     const char* instance(int index) const;
     const char* pin(int index) const;
     int pinIsMustJoin(int index) const;
     int pinIsSynthesized(int index) const;
     int numSubnets() const;
     defiSubnet* subnet(int index);

     int isFixed() const;
     int isRouted() const;
     int isCover() const;

     int numWires() const;
     defiWire* wire(int index);

     int numVpins() const;
     defiVpin* vpin(int index) const;

     int hasProps() const;
```

```
int hasWeight() const;
int hasSubnets() const;
int hasSource() const;
int hasFixedbump() const;
int hasFrequency() const;
int hasPattern() const;
int hasOriginal() const;
int hasCap() const;
int hasUse() const;
int hasStyle() const;
int hasNonDefaultRule() const;
int hasVoltage() const;
int hasSpacingRules() const;
int hasWidthRules() const;
int hasXTalk() const;

int numSpacingRules() const;
void spacingRule(int index, char** layer, double* dist,
     double* left, double* right);
int numWidthRules() const;
void widthRule(int index, char** layer, double* dist);
double voltage() const;

int XTalk() const;
const char* source() const;
double frequency() const;
const char* original() const;
const char* pattern() const;
double cap() const;
const char* use() const;
int style() const;
const char* nonDefaultRule() const;

int numPaths() const;
defiPath* path(int index);

int numShields() const;
defiShield* shield(int index);
int numShieldNets() const;
const char* shieldNet(int index) const;
int numNoShields() const;
defiShield* noShield(int index);

int numPolygons() const;
const char* polygonName(int index) const;
struct defiPoints getPolygon(int index) const;
int numRectangles() const;
const char* rectName(int index) const;
int xl(int index) const;
int yl(int index) const;
int xh(int index) const;
```

```
     int yh(int index) const;
     int polyMask(int index) const;
     int rectMask(int index) const;
     int topMaskNum(int index) const;
     int cutMaskNum(int index) const;
     int bottomMask(int index) const;}
```

Examples

The following example shows a callback routine with the type `defrSNetCbkType`. Callback routines for the type `defrSNetStartCbkType` and `defrSNetEndCbkType` are similar to the example for `defrViaStartCbkType` and `defrViaEndCbkType` in the Via section. This example only shows how to retrieve part of the data from the `defiNet` class.

```
int snetCB (defrCallbackType_e type,
                 defiNet* snetInfo,
                 defiUserData userData) {

     int         i, x, y, newLayer;
     char*       layerName;
     double      dist, left, right;
     defiPath*   p;
     int         path;
     defiShield* shield;

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

     // compName & pinName
         for (i = 0; i < net->numConnections(); i++)
             printf ("( %s %s )\n", net->instance(i), net->pin(i));

     // specialWiring
         if (net->isFixed()) {
             printf("FIXED\n");
         }

         if (net->numPaths()) {
             newLayer = 0;
             for (i = 0; i < net->numPaths(); i++) {
                 p = net->path(i);
                 p->initTraverse();
                 while ((path = (int)p->next()) != DEFIPATH_DONE) {
                     switch (path) {
                         case DEFIPATH_LAYER:
```

```
                    if (newLayer == 0) {
                        printf("%s ", p->getLayer());
                        newLayer = 1;
                    } else
                        printf("NEW %s ", p->getLayer());
                    break;

                case DEFIPATH_VIA:
                    printf("%s ", p->getVia());
                    break;

                case DEFIPATH_WIDTH:
                    printf("%d ", p->getWidth());
                    break;

                case DEFIPATH_POINT:
                    p->getPoint(&x, &y);
                    printf("( %d %d ) ", x, y);
                    break;

                case DEFIPATH_TAPER:
                    printf("TAPER ");
                    break;

                case DEFIPATH_SHAPE:
                    printf(" SHAPE %s ", p->getShape());
                    break;
            }
        }
        printf("\n");
    }
}
// SHIELD
// testing the SHIELD for 5.3
if (net->numShields()) {
    for (i = 0; i < net->numShields(); i++) {
        shield = net->shield(i);
        printf("\n+ SHIELD %s ",
                shield->defiShield::shieldName());
        newLayer = 0;
        for (j = 0; j < shield->defiShield::numPaths(); j++) {
            p = shield->defiShield::path(j);
            p->initTraverse();
            while ((path = (int)p->next()) != DEFIPATH_DONE) {
                switch (path) {
                    case DEFIPATH_LAYER:
                        if (newLayer == 0) {
                            printf("%s ", p->getLayer());
                            newLayer = 1;
                        } else
                            printf("NEW %s ", p->getLayer());
                        break;
```

```
                        case DEFIPATH_VIA:
                            printf("%s ", p->getVia());
                           break;

                        case DEFIPATH_WIDTH:
                            printf("%d ", p->getWidth());
                           break;

                        case DEFIPATH_POINT:
                            p->getPoint(&x, &y);
                            printf("( %d %d ) ", x, y);
                           break;

                        case DEFIPATH_TAPER:
                            printf("TAPER ");
                           break;

                    }
                }
                printf("\n");
            }
        }
    }
    // layerName spacing

    if (net->hasSpacingRules()) {
        for (i = 0; i < net->numSpacingRules(); i++) {
            net->spacingRule(i, &layerName, &dist, &left, &right);
            if (left == right)
                printf("SPACING %s %g\n", layerName, dist);
            else
                printf("SPACING %s %g RANGE %g %g\n",
                        layerName, dist, left, right);
        }
    }
    return 0;
}
```

defiNonDefault

Retrieves data from the `NONDEFAULTRULES` statement in the DEF file. For syntax information about the DEF `NONDEFAULTRULES` statement, see "Nondefault Rules," in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiNonDefault {
     const char* name() const;
     int hasHardspacing() 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 hasLayerDiagWidth(int index) const;
int hasLayerSpacing(int index) const;
int hasLayerWireExt(int index) const;
int numVias() const;
const char* viaName(int index) const;
int numViaRules() const;
const char* viaRuleName(int index) const;
int hasMinCuts() const;
void minCuts(const char **cutLayerName, int *numCuts) const;}
```

defiOrdered

Retrieves data from the `ORDERED` statement in the `SCANCHAINS` statement of the DEF file. For syntax information about the DEF `SCANCHAINS` statement, see "Scan Chains" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiOrdered {
    int num() const;
    char** inst() const;
    char** in() const;
    char** out() const;
    int* bits() const; }
```

defiPath

Retrieves data from the *regularWiring* and *specialWiring* specifications in the `NETS` and `SPECIALNETS` sections of the DEF file. For syntax information about the DEF `SPECIALNETS` and `NETS` statements, see "Special Nets" and "Nets" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiPath {
     void initTraverse();
     void initTraverseBackwards();
     int next();
     int prev();
     const char* getLayer(); .
     const char* getTaperRule();
     const char* getVia();
     const char* getShape();
     int getStyle();
     int getViaRotation();
     const char* getViaRotationStr();
     void getViaData(int* numX, int* numY, int* stepX, int* stepY);
     int getWidth();
     void getPoint(int* x, int* y);
     void getFlushPoint(int* x, int* y, int* ext);
     int getMask();
     int getViaTopMask();
     int getViaCutMask();
     int getViaBottomMask();
     int getRectMask();}
```

Examples

For a `defiPath` example, see the example in the `defiNet` section.

defiPin

Retrieves data from the `PINS` statement in the DEF file. For syntax information about the DEF `PINS` statement, see “Pins” in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiPin {
     const char* pinName() const;
     const char* netName() const;

     int hasDirection() const;
     int hasUse() const;
     int hasLayer() const;
     int hasPlacement() const;
     int isUnplaced() const;
     int isPlaced() const;
     int isCover() const;
     int isFixed() const;
```

```
int placementX() const;
int placementY() const;
const char* direction() const;
const char* use() const;
int numLayer() const;
const char* layer(int index) const;
void bounds(int index, int* xl, int* yl, int* xh, int* yh) const;
int hasLayerSpacing(int index) const;
int hasLayerDesignRuleWidth(int index) const;
int layerSpacing(int index) const;
int layerDesignRuleWidth(int index) const;
int numPolygons() const;
const char* polygonName(int index) const;
struct defiPoints getPolygon(int index) const;
int hasPolygonSpacing(int index) const;
int hasPolygonDesignRuleWidth(int index) const;
int polygonSpacing(int index) const;
int polygonDesignRuleWidth(int index) const;
int hasNetExpr() const;
int hasSupplySensitivity() const;
int hasGroundSensitivity() const;
const char* netExpr() const;
const char* supplySensitivity() const;
const char* groundSensitivity() const;
int orient() const;                    // optional- For information, see
                                       //"Orientation Codes" on page 19
const char* orientStr() const;
int hasSpecial() const;

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

int hasAPinPartialMetalArea() const;
int numAPinPartialMetalArea() const;
int APinPartialMetalArea(int index) const;
int hasAPinPartialMetalAreaLayer(int index) const;
const char* APinPartialMetalAreaLayer(int index) const;

int hasAPinPartialMetalSideArea() const;
int numAPinPartialMetalSideArea() const;
int APinPartialMetalSideArea(int index) const;
int hasAPin PartialMetalSideAreaLayer(int index) const;
const char* APinPartialMetalSideAreaLayer(int index) const;

int hasAPinDiffArea() const;
int numAPinDiffArea() const;
int APinDiffArea(int index) const;
int hasAPinDiffAreaLayer(int index) const;
const char* APinDiffAreaLayer(int index) const;
```

```
int hasAPinPartialCutArea() const;
int numAPinPartialCutArea() const;
int APinPartialCutArea(int index) const;
int hadAPinPartialCutAreaLayer(int index) const;
const char* APinPartialCutAreaLayer(int index) const;

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

int  hasPort() const;
int  numPorts() const;
defiPinPort* pinPort(int index) const;
int layerMask(int index) const;
int polygonMask(int index) const;
int viaTopMask(int index) const;
int viaCutMask(int index) const;
int viaBottomMask(int index) const;}
```

Examples

The following example shows a callback routine with the type `defrPinCbkType`. Callback routines for the type `defrStartPinsCbkType` and `defrPinEndCbkType` are similar to the example for `defrViaStartCbkType` and `defrViaEndCbkType` in the Via section.

```
int pinCB (defrCallbackType_e type,
                defiPin* pinInfo,
                defiUserData userData) {

     int i;

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

          printf("%s NET %s\n", pinInfo->pinName(),
                 pinInfo->netName());
         if (pinInfo->hasDirection())
             printf("  DIRECTION %s\n", pinInfo->direction());
         if (pinInfo->hasUse())
             printf("  USE %s\n", pinInfo->use());
         if (pinInfo->hasLayer()) {
             printf("  LAYER %s ", pinInfo->layer());
             pinInfo->bounds(&xl, &yl, &xh, &yh);
             printf("%d %d %d %d\n", xl, yl, xh, yh);
         }

          if (pinInfo->hasPlacement()) {
```

```
            if (pinInfo->isPlaced())
                printf("  PLACED\n");
            if (pinInfo->isCover())
                printf("  COVER\n");
            if (pinInfo->isFixed())
                printf("  FIXED\n");
            printf("( %d %d ) %d ", pinInfo->placementX(),
                   pinInfo->placementY(),
                   pinInfo->orient());
        }
        if (pinInfo->hasSpecial())
            printf("  SPECIAL\n");
        return 0;}
```

defiPinAntennaModel

Retrieves antenna model information in the `PINS` statement in the DEF file. For syntax information about the DEF `PINS` statement, see <u>“Pins”</u> in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiPinAntennaModel {
     char* antennaOxide() const;

     int hasAPinGateArea() const;
     int numAPinGateArea() const;
     int APinGateArea(int index) const;
     int hasAPinGateAreaLayer(int index) const;
     const char* APinGateAreaLayer(int index) const;

     int hasAPinMaxAreaCar() const;
     int numAPinMaxAreaCar() const;
     int APinMaxAreaCar(int index) const;
     int hasAPinMaxAreaCarLayer(int index) const;
     const char* APinMaxAreaCarLayer(int index) const;

     int hasAPinMaxSideAreaCar() const;
     int numAPinMaxSideAreaCar() const;
     int APinMaxSideAreaCar(int index) const;
     int hasAPinMaxSideAreaCarLayer(int index) const;
     const char* APinMaxSideAreaCarLayer(int index) const;

     int hasAPinMaxCutCar() const;
     int numAPinMaxCutCar() const;
     int APinMaxCutCar(int index) const;
     int hasAPinMaxCutCarLayer(int index) const;
     const char* APinMaxCutCarLayer(int index) const; }
```

defiPinPort

Retrieves data from the PINS PORT statement in the DEF file. For syntax information about the DEF `PINS PORT` statement, see "Pins" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiPinPort {
     int numLayer() const;
     const char* layer(int index) const;
     int hasLayerSpacing(int index) const;
     int hasLayerDesignRuleWidth(int index) const;
     int layerSpacing(int index) const;
     int layerDesignRuleWidth(int index) const;
     int numPolygons() const;
     const char* polygonName(int index) const;
     struct defiPoints getPolygon(int index) const;
     int hasPolygonSpacing(int index) const;
     int hasPolygonDesignRuleWidth(int index) const;
     int polygonSpacing(int index) const;
     int polygonDesignRuleWidth(int index) const;
     int numVias() const;
     const char* viaName(int index) const;
     int viaPtX (int index) const;
     int viaPtY (int index) const;
     int hasPlacement() const;
     int isPlaced() const;
     int isCover() const;
     int isFixed() const;
     int placementX() const;
     int placementY() const;
     int orient() const;
     const char* orientStr() const;

     int layerMask(int index) const;
     int polygonMask(int index) const;
     int viaTopMask(int index) const;
     int viaCutMask(int index) const;
     int viaBottomMask(int index) const;};}
```

defiPinProp

Retrieves data from the `PINPROPERTIES` statement in the DEF file. For syntax information about the DEF `PINPROPERTIES` statement, see "Pin Properties" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiPinProp {
     int isPin() const;
     const char* instName() const;
     const char* pinName() 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); }
```

Examples

The following example shows a callback routine with the type `defrPinPropCbkType`. Callback routines for the type `defrPinPropStartCbkType` and `defrPinPropEndCbkType` are similar to the example for `defrViaStartCbkType` and `defrViaEndCbkType` in the Via section.

```
int pinpropCB (defrCallbackType_e type,
                    defiPinProp* pinpropInfo,
                    defiUserData userData) {
         int i;

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

     if (pinpropInfo->isPin())
            printf("PIN %s\n", pinpropInfo->pinName());
         else
            printf("%s %s\n", pinpropInfo->instName(),
                    pinpropInfo->pinName());
         if (pinpropInfo->numProps() > 0) {
            for (i = 0; i < pinpropInfo->numProps(); i++) {
                printf(" PROPERTY %s %s\n", pinpropInfo->propName(i),
                        pinpropInfo->propValue(i));
            }
         }

     return 0;}
```

defiPoints

Retrieves a list of points for polygons in the DEF file.

C++ Syntax

```
struct defiPoints {
     int numPoints;
     int* x;
     int* y;}
```

defiProp

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

The string of the property is returned by the C++ function `string` or the C function `defiProp_string`. A property can have a number and a range, which are returned by the function `hasNumber` and `hasRange`. The actual values are returned by the functions `number`, `left`, and `right`.

C++ Syntax

```
class defiProp {
     const char* string() const;
     const char* propType() const;
     const char* propName() const;
     char  dataType() const;         // either I:integer, R:real, S:string,
                                     // Q:quotestring, or N:nameMapString
     int hasNumber() const;
     int hasRange() const;
     int hasString() const;
     int hasNameMapString() const;
     double number() const;
     double left() const;
     double right() const;}
```

Examples

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

```
int propDefStartCB (defrCallbackType_e type,
                         void* dummy,
                        defiUserData userData) {

         // Check if the type is correct
         if (type != defrPropDefStartCbkType) {
             printf("Type is not defrPropDefStartCbkType,
                     terminate parsing.\n");
             return 1;
         }
         printf("PROPERTYDEFINITIONS\n");
         return 0;}
```

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

```
int propDefCB (defrCallbackType_e type,
                    defiProp* propInfo,
                    defiUserData userData) {
         // Check if the type is correct
         if (type != defrPropCbkType) {
             printf("Type is not defrPropCbkType, terminate
                     parsing.\n");
             return 1;
         }

     // Check the object type of the property definition
         if (strcmp(propInfo->propType(), "design") == 0)
             printf("DESIGN %s ", propInfo->propName());
         else if (strcmp(propInfo->propType(), "net") == 0)
             printf("NET %s ", propInfo->propName());
         else if (strcmp(propInfo->propType(), "component") == 0)
             printf("COMPONENT %s ", propInfo->propName());
         else if (strcmp(propInfo->propType(), "specialnet") == 0)
             printf("SPECIALNET %s ", propInfo->propName());
         else if (strcmp(propInfo->propType(), "group") == 0)
             printf("GROUP %s ", propInfo->propName());
         else if (strcmp(propInfo->propType(), "row") == 0)
             printf("ROW %s ", propInfo->propName());
         else if (strcmp(propInfo->propType(), "componentpin") == 0)
             printf("COMPONENTPIN %s ", propInfo->propName());
         else if (strcmp(propInfo->propType(), "region") == 0)
             printf("REGION %s ", propInfo->propName());
         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 `defrPropDefEndCbkType`, and `void *`. This callback routine marks the end of the Property Definitions section.

```
int propDefEndCB (defrCallbackType_e type,
                       void* dummy,
                       defiUserData userData) {
        // Check if the type is correct
        if (type != defrPropDefEndCbkType) {
            printf("Type is not defrPropDefEndCbkType,
                       terminate parsing.\n");
            return 1;
    }
```

defiRegion

Retrieves data from the `REGIONS` statement in the DEF file. For syntax information about the DEF `REGIONS` statement, see "Regions" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiRegion {
     const char* name() 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 hasType() const;
     const char* type() const;

     int numRectangles() const;
     int xl(int index) const;
```

```
     int yl(int index) const;
     int xh(int index) const;
     int yh(int index) const;}
```

Examples

The following example shows a callback routine with the type `defrRegionCbkType`. Callback routines for the type `defrRegionStartCbkType` and `defrRegionEndCbkType` are similar to the example for `defrViaStartCbkType` and `defrViaEndCbkType` in the Via section.

```
int regionCB (defrCallbackType_e type,
                   defiRegion* regionInfo,
                   defiUserData userData) {
         int i;
         char* name;

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

         for (i = 0; i < regionInfo->numRectangles(); i++)
             printf("%d %d %d %d \n", regionInfo->xl(i),
                     regionInfo->yl(i), regionInfo->xh(i),
                    regionInfo->yh(i));
     return 0;}
```

defiRow

Retrieves data from the `ROW` statement in the DEF file. For syntax information about the DEF `ROW` statement, see "Rows" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiRow {
     const char* name() const;
     const char* macro() const;
     double x() const;
     double y() const;
     int orient() const;                    // optional-For information,see
                                            //"Orientation Codes" on page 19
     const char* orientStr() const;
     int hasDo() const;
     double xNum() const;
```

```
     double yNum() const;
     int hasDoStep() const;
     double xStep() const;
     double yStep() 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;}
```

Examples

The following example shows a die area routine using a callback routine with the type `defrDieAreaCbkType`, and the class `defiRow`.

```
int diearea (defrCallbackType_e type,
                defiRow* dieareaInfo,
                defiUserData userData) {

     // Check if the type is correct
         if (type != defrDieAreaCbkType) {
             printf("Type is not defrDieAreaCbkType, terminate
                     parsing.\n");
             return 1;
         }
         printf("DIEAREA %d %d %d %d\n", diearea->xl(), diearea->yl(),
                diearea->xh(), diearea->yh());
         return 0;}
```

The following example shows a row routine using a callback routine with the type `defrRowCbkType`, and the class `defiRow`.

```
int rowCB (defrCallbackType_e type,
                defiRow* rowInfo,
                defiUserData userData) {

     int i;

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

```
     printf("ROW %s %s %g %g %d ", rowInfo->name(),
            rowInfo->macro(), rowInfo->x(), rowInfo->y(),
            rowInfo->orient());
     printf("DO %g BY %g STEP %g %g\n", rowInfo->xNum(),
            rowInfo->yNum(),rowInfo->xStep(), row->yStep());
     if (rowInfo->numProps() > 0) {
        for (i = 0; i < rowInfo->numProps(); i++) {
            printf("   PROPERTY %s %s\n", rowInfo->propName(i),
                   rowInfo->propValue(i));
        }
    }
    return 0;}
```

defiScanchain

Retrieves data from the `SCANCHAINS` statement in the DEF file. For syntax information about the DEF `SCANCHAINS` statement, see <u>"Scan Chains"</u> in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiScanchain {
     const char* name() const;
     int hasStart() const;
     int hasStop() const;
     int hasFloating() const;
     int hasOrdered() const;
     int hasCommonInPin() const;
     int hasCommonOutPin() const;
     int hasPartition() const;
     int hasPartitionMaxBits() const;

     void start(char** inst, char** pin) const;
     void stop(char** inst, char** pin) const;

     int numOrdered() const;

     void ordered(int index, int* size, char*** inst, char*** inPin,
          char*** outPin, int** bits) const;
     void floating(int* size, char*** inst, char*** inPin,
          char*** outPin, int** bits) const;

     const char* commonInPin() const;
     const char* commonOutPin() const;

     const char* partitionName() const;
     int partitionMaxBits(); }
```

Examples

The following example shows a callback routine with the type `defrScanchainCbkType`. Callback routines for the type `defrScanchainsStartCbkType` and `defrScanchainsEndCbkType` are similar to the example for `defrViaStartCbkType` and `defrViaEndCbkType` in the Via section.

```
int scanchainCB (defrCallbackType_e type,
                      defiScanchain* scanchainInfo,
                      defiUserData userData) {

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

         printf("%s\n", scanchainInfo->name());
         if (scanchainInfo->hasStart()) {
             scanchainInfo->start(&a1, &b1);
             printf("   START %s %s\n", a1, b1);
         }
         if (scanchainInfo->hasStop()) {
             scanchainInfo->stop(&a1, &b1);
             printf("   STOP %s %s\n", a1, b1);
         }
         if (scanchainInfo->hasCommonInPin() ||
             scanchainInfo->hasCommonOutPin()) {
             printf("   COMMONSCANPINS ");
             if (scanchainInfo->hasCommonInPin())
                printf(" ( IN %s ) ", scanchainInfo->commonInPin());
             if (scanchainInfo->hasCommonOutPin())
                printf(" ( OUT %s ) ",scanchainInfo->commonOutPin());
             printf("\n");
         }
         if (scanchainInfo->hasFloating()) {
             scanchainInfo->floating(&size, &inst, &inPin, &outPin);
             if (size > 0)
                 printf("  + FLOATING\n");
             for (i = 0; i < size; i++) {
                 printf("    %s ", inst[i]);
                 if (inPin[i])
                    printf("( IN %s ) ", inPin[i]);
                 if (outPin[i])
                    printf("( OUT %s ) ", outPin[i]);
                 printf("\n");
             }
             printf("\n");
         }
```

```
if (scanchainInfo->hasOrdered()) {
    for (i = 0; i < scanchainInfo->numOrderedLists(); i++) {
        scanchainInfo->ordered(i, &size, &inst, &inPin,
        &outPin);
        if (size > 0)
            printf("  + ORDERED\n");
        for (i = 0; i < size; i++) {
            printf("    %s ", inst[i]);
            if (inPin[i])
               printf("( IN %s ) ", inPin[i]);
            if (outPin[i])
               printf("( OUT %s ) ", outPin[i]);
            printf("\n");
        }
    }
    printf("\n");
}
return 0;}
```

defiShield

Retrieves data from the `SPECIALNETS` statement in the DEF file. For syntax information about the DEF `SPECIALNETS` statement, see "Special Nets" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiShield {
     const char* shieldName() const;
     int numPaths() const;
     defiPath* path(int index);}
```

Examples

For a `defiShield` example, see the example in the `defiNet` section.

defiSite

Retrieves data from any obsolete `SITE` sections of the DEF file.

C++ Syntax

```
class defiSite {
     double x_num() const;
     double y_num() const;
     double x_step() const;
     double y_step() const;
     double x_orig() const;
     double y_orig() const;
     int orient() const;              // optional- For information, see
                                      //"Orientation Codes" on page 19
     const char* orientStr() const;
     const char* name() const;}
```

Examples

The following example shows a callback routine with the type `defrCanplaceCbk` and `defrCannotOccupyCbk`.

```
int siteCB (defrCallbackType_e type,
                 defiSite siteInfo,
                 defiUserData userData) {

     // Check if the type is correct
         if ((type != defrCanplaceCbk) && (type !=
              defrCannotOccupyCbk)) {
              printf("Type is not defrCanplaceCbk and not
                     defrCannotOccupyCbk,\n");

              printf("terminate parsing.\n");
              return 1;
         }

         printf("CANPLACE %s %g %g %s ", siteInfo->name(),
                 siteInfo->x_orig(), siteInfo->y_orig(),
                 orientStr(siteInfo->orient()));
         printf("DO %d BY %d STEP %g %g ;\n", siteInfo->x_num(),
                 siteInfo->y_num(),
                siteInfo->x_step(), siteInfo->y_step());
         return 0;}
```

defiSlot

Retrieves data from the `SLOTS` statement in the DEF file. For syntax information about the DEF `SLOTS` statement, see "Slots" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiSlot {
     int hasLayer() const;
     const char* layerName() const;

     int numRectangles() const;
     int xl(int index) const;
     int yl(int index) const;
     int xh(int index) const;
     int yh(int index) const;

     int numPolygons() const;
     struct defiPoints getPolygon(int index) const;}
```

defiStyles

Retrieves data from the `STYLES` statement in the DEF file. For syntax information about the DEF `STYLES` statement, see "Styles," in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiStyles {
     int style() const;
     struct defiPoints getPolygon() const;}
```

defiSubnet

Retrieves data from the `SUBNETS` statement in the `NETS` statement in the DEF file. For syntax information about the DEF `NETS` statement, see "Nets" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiSubnet {
     const char* name() const;
     int numConnections();
     const char* instance(int index);
     const char* pin(int index);
     int pinIsSynthesized(int index);
     int pinIsMustJoin(int index);
     int isFixed() const;
     int isRouted() const;
     int isCover() const;
     int hasNonDefaultRule() const;
```

```
     int hasShield() const;
     int hasShieldNet() const;
     int hasNoShieldNet() const;
     int numPaths() const;
     defiPath* path(int index);
     const char* nonDefaultRule() const;
     int numWires() const;
     defiWire* wire(int index);}
```

defiTrack

Retrieves data from the TRACKS statement in the DEF file. For syntax information about the DEF TRACKS statement, see "Tracks" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiTrack {
     const char* macro() const;
     double x() const;
     double xNum() const;
     double xStep() const;
     int numLayers() const;
     const char* layer(int index) const;
     int firstTrackMask() const;
     int sameMask() const;}
```

Examples

The following example shows a callback routine with the type defrTrackCbkType, and the class defiTrack.

```
int trackCB (defrCallbackType_e type,
                  defiTrack* trackInfo,
                  defiUserData userData) {
         int i;

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

     printf("TRACKS %s %g DO %g STEP %g LAYER ",
                 trackInfo->macro(),
                trackInfo->x(), trackInfo->xNum(), trackInfo->xStep());
     for (i = 0; i < trackInfo->numLayers(); i++)
```

```
            printf("%s ", trackInfo->layer(i));
        printf("\n");

    return 0;}
```

defiVia

Retrieves data from the `VIAS` statement in the DEF file. For syntax information about the DEF `VIAS` statement, see "Vias" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiVia {
    const char* name() const;
    const char* pattern() const;
    int hasPattern() const;
    int numLayers() const;
    void layer(int index, char** layer, int* xl, int* yl,
        int* xh, int* yh) const;
    int numPolygons() const;
    const char* polygonName(int index) const;
    struct defiPoints getPolygon(int index) const:
    int hasViaRule() const;
    void viaRule(char** viaRuleName, int* xSize, int* ySize,
        char** botLayer, char** cutLayer, char** topLayer,
        int* xCutSpacing, int* yCutSpacing, int* xBotEnc, int* yBotEnc,
        int* xTopEnc, int* yTopEnc) const;
    int hasRowCol() const;
    void rowCol(int* numCutRows, int* numCutCols) const;
    int hasOrigin() const;
    void origin(int* xOffset, int* yOffset) const;
    int hasOffset() const;
    void offset(int* xBotOffset, int* yBotOffset, int* xTopOffset
        int* yTopOffset) const;
    int hasCutPattern() const;
    const char* cutPattern() const;
    int rectMask(int index) const;
    int polyMask(int index) const; }
```

Examples

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

```
int viaStartCB (defrCallbackType_e type,
                int numVias,
                defiUserData userData) {
```

```
    // Check if the type is correct
    if ((type != defrViaStartCbkType)) {
        printf("Type is not defrViaStartCbkType terminate
            parsing.\n");
        return 1;
    }
    printf("VIAS %d\n", numVias);
    return 0;}
```

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

```
int viaCB (defrCallbackType_e type,
                defiVia* viaInfo,
                defiUserData userData) {
    int i, xl, yl, xh, yh;
    char* name;

    // Check if the type is correct
        if ((type != defrViaCbkType)) {
            printf("Type is not defrViaCbkType terminate parsing.\n");
            return 1;
        }
    printf("Via name is %s ", viaInfo->name());
    if (viaInfo->hasPattern())
            printf("  PATTERNNAME %s\n", viaInfo->pattern());
    for (i = 0; i < viaInfo->numLayers(); i++) {
            viaInfo->layer(i, &name, &xl, &yl, &xh, &yh);
            printf(" RECT %s %d %d %d %d \n", name, xl, yl, xh, yh);
        }

   return 0;}
```

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

```
int viaEndCB (defrCallbackType_e type,
                   void* ptr,
                   defiUserData userData) {

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

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

defiViaData

Retrieves via array data from the `SPECIALNETS` statement in the DEF file. For syntax information about the DEF `SPECIALNETS` statement, see "Special Nets" in the *LEF/DEF Language Reference*.

C++ Syntax

```
struct defiViaData {
     int numX;
     int numY;
     int stepX;
     int stepY;}
```

defiVpin

Retrieves data from the `VPIN` statement in the `NETS` statement in the DEF file. For syntax information about the DEF `NETS` statement, see "Nets" and in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiVpin {
     int xl() const;
     int yl() const;
     int xh() const;
     int yh() const;
     char status() const;
     int orient() const;
     const char* orientStr() const;
     int xLoc() const;
     int yLoc() const;
     const char* name() const;
     const char* layer() const;}
```

defiWire

Retrieves data from the *`regularWiring`* or *`specialWiring`* section of the `NETS` or `SPECIALNETS` statements in the DEF file. For syntax information about the DEF `NETS` and `SPECIALNETS` statements, see "Nets" and "Special Nets" in the *LEF/DEF Language Reference*.

C++ Syntax

```
class defiWire {
     const char* wireType() const;
     const char* wireShieldNetName() const;
     int numPaths() const;
     defiPath* path(int index);}
```

5

DEF Writer Callback Routines

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

When you use callback routines, the writer creates a DEF 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	yes	yes
Bus Bit Characters	yes	yes
Divider	yes	yes
Design	yes	no
Technology	no	no
Units	no	no
History	no	no
Property Definition	no	no
Die Area	no	no
Rows	no	no
Tracks	no	no
Gcell Grid	no	no
Vias	no	no
Regions	no	no
Components	yes	no

Section	Required	Default Available
Pins	no	no
Pin Properties	no	no
Special Nets	no	no
Nets	yes	no
Scan chains	no	no
Groups	no	no
Extensions	no	no
Design End	yes	no

Callback Function Format

All callback functions use the following format.

```
int UserCallbackFunctions(
     defwCallbackType_e callBackType,
     defiUserData 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 DEF data.

User Data

The data argument is a four-byte data item that you set. The DEF 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 DEF writer callback-setting routines and the associated callback types.

DEF Information	Setting Routine	Callback Types
Blockages	`void defwSetBlockageCbk (defwVoidCbkFnType)`	`defwBlockageCbkType`
Bus Bit Characters	`void defwSetBusBitCbk (defwVoidCbkFnType)`	`defwBusBitCbkType`
Components	`void defwSetComponentCbk (defwVoidCbkFnType)`	`defwComponentCbkType`
Design	`void defwSetDesignCbk (defwVoidCbkFnType)`	`defwDesignCbkType`
Design End	`void defwSetDesignEndCbk (defwVoidCbkFnType)`	`defwDesignEndCbkType`
Die Area	`void defwSetDieAreaCbk (defwVoidCbkFnType)`	`defwDieAreaCbkType`
Divider	`void defwSetDividerCbk (defwVoidCbkFnType)`	`defwDividerCbkType`
Extensions	`void defwSetExtCbk (defwVoidCbkFnType)`	`defwExtCbkType`
Gcell Grid	`void defwSetGcellGridCbk (defwVoidCbkFnType)`	`defwGcellGridCbkType`
Groups	`void defwSetGroupCbk (defwVoidFnType)`	`defwGroupCbkType`
History	`void defwSetHistoryCbk (defwVoidCbkFnType)`	`defwHistoryCbkType`
Nets	`void defwSetNetCbk (defwVoidCbkFnType)`	`defwNetCbkType`
Pins	`void defwSetPinCbk (defwVoidCbkFnType)`	`defwPinCbkType`
Pin Properties	`void defwSetPinPropCbk (defwVoidCbkFnType)`	`defwPinPropCbkType`

DEF Information	Setting Routine	Callback Types
Property Definitions	`void defwSetPropDefCbk (defwVoidCbkFnType)`	`defwPropDefCbkType`
Regions	`void defwSetRegionCbk (defwVoidCbkFnType)`	`defwRegionCbkType`
Rows	`void defwSetRowCbk (defwVoidCbkFnType)`	`defwRowCbkType`
Special Nets	`void defwSetSNetCbk (defwVoidCbkFnType)`	`defwSNetCbkType`
Scan Chains	`void defwSetScanchainCbk (defwVoidCbkFnType)`	`defwScanchainCbkType`
Technology	`void defwSetTechnologyCbk (defwVoidCbkFnType)`	`defwTechCbkType`
Tracks	`void defwSetTrackCbk (defwVoidCbkFnType)`	`defwTrackCbkType`
Units	`void defwSetUnitsCbk (defwVoidCbkFnType)`	`defwUnitsCbkType`
Version	`void defwSetVersionCbk (defwVoidCbkFnType)`	`defwVersionCbkType`
Vias	`void defwSetViaCbk (defwVoidCbkFnType)`	`defwViaCbkType`

6

DEF Writer Routines

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

Routines	DEF File Section
DEF Writer Setup and Control	Initialization and global variables
Blockages	`BLOCKAGES` statement
Bus Bit Characters	`BUSBITCHARS` statement
Components	`COMPONENTS` statement
Design Name	`DESIGN` statement
Die Area	`DIEAREA` statement
Divider Character	`DIVIDERCHAR` statement
Extensions	`EXTENSIONS` statement
Fills	`FILLS` statement
GCell Grid	`GCELLGRID` statement
Groups	`GROUPS` statement
History	`HISTORY` statement
Nets	`NETS` statement
Regular Wiring	*regularWiring* statement in a `NETS` statement
Subnet	`SUBNET` statement in a `NETS` statement
Nondefault Rules	`NONDEFAULTRULES` statement
Pins	`PINS` statement
Pin Properties	`PINPROPERTIES` statement

Routines	**DEF File Section**
Property Definitions	`PROPERTYDEFINITIONS` statement
Property Statements	`PROPERTY` statements
Regions	`REGIONS` statement
Rows	`ROW` statement
Special Nets	`SPECIALNETS` statement
Special Wiring	*specialWiring* statement in a `SPECIALNETS` statement
Shielded Routing	*shielded routin*g statement in a `SPECIALNETS` statement
Scan Chains	`SCANCHAINS` statement
Slots	`SLOTS` statement
Styles	`STYLES` statement
Technology	`TECHNOLOGY` statement
Tracks	`TRACKS` statement
Units	`UNITS` statement
Version	`VERSION` statement
Vias	`VIAS` statement

DEF Writer Setup and Control

The DEF writer setup and control routines initialize the reader and set global variables that are used by the DEF file. You must begin a DEF file with either the `defwInit` routine or the `defwInitCbk` routine. You must end a DEF file with the `defwEnd` routine. All other routines must be used between these routines. The remaining routines described in this section are provided as utilities.

For an example on how to set up the writer, see "Setup Examples" on page 110.

All routines return `0` if successful.

defwInit

Initializes the DEF writer. Use this routine if you do not want to use the callback mechanism.

Syntax

```
int defwInit (
     FILE* file,
     int vers1,
     int vers2,
     const char* caseSensitive,
     const char* dividerChar,
     const char* busBitChars,
     const char* designName,
     const char* technology,
     const char* array,
     const char* floorplan,
     double units)
```

Arguments

file

Specifies the name of the DEF file to create.

vers1, vers2

Specifies which version of LEF/DEF is being used. *vers1* specifies the major number. *vers2* specifies the minor number.

caseSensitive

Note: The NAMECASESENSITIVE statement is obsolete; therefore the writer ignores this argument.

dividerChar

Writes the DIVIDERCHAR statement that specifies the character used to express hierarchy when DEF names are mapped to or from other databases. The character must be enclosed in double quotation marks.

busBitChars

Writes the BUSBITCHARS statement that specifies the pair of characters used to specify bus bits when DEF names are mapped to or from other databases. The characters must be enclosed in double quotation marks.

designName

Writes the DESIGN statement that specifies a name for the design.

technology

Writes the TECHNOLOGY statement that specifies a technology name for the design.

units

Writes the `UNITS` statement that specifies how to convert DEF units.

defwInitCbk

Also initializes the DEF writer. Use this routine if you want to use the callback mechanism. If you use this routine, you must also use the following routines:

- `defwVersion`
- `defwBusBitChars`
- `defwDividerChar`
- `defwDesignName`

If you do not include these routines, default values are used.

Syntax

```
int defwInit(
     FILE* file);
```

Arguments

file

Specifies the name of the DEF file to create.

defwEnd

Ends the DEF file. This routine is required and must be used last.

Syntax

```
int defwEnd(void)
```

defwCurrentLineNumber

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

Syntax

```
int defwCurrentLineNumber(void)
```

defwNewLine

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

Syntax

```
int defwNewLine()
```

defwAddComment

Allows you to enter any comment into the DEF file. This statement automatically adds a pound symbol (#) to the beginning of the comment statement.

Syntax

```
int defwAddComment(
     const char* comment)
```

defwAddIntent

Automatically indents a statement by adding three blank spaces to the beginning of the statement. This routine does not require any arguments.

Syntax

```
int defwAddIndent()
```

defwPrintError

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

Syntax

```
void defwPrintError(
     int status)
```

Arguments

status

Specifies the nonzero integer returned by the DEF writer routines.

Setup Examples

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

- You call the write routines in your own sequence. The writer makes sure that some routines are called before others, but it is mainly your responsibility to make sure the sequence is correct, and all the 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 will issue a warning. If there is a default routine, the writer will invoke 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 def file for the writer to write
     if ((f = fopen("defOutputFileName","w")) == 0) {
          printf("Couldn't open output file '%s'\n",
                "defOutputFileName");
     return(2);
     }


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


    res = defwEnd();
         ...
```

```
    fclose(f);

     return 0;
     }
```

The following example uses the writer with callbacks.

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


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


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


    res = defwEncrypt();  // Set flag to write in encrypted format
     ...


    // Set the user callback routines
     defwSetArrayCbk (arrayCB);
     defwSetBusBitCbk (busbitCB);
     defwSetCaseSensitiveCbk (casesensitiveCB);
     defwSetComponentCbk (componentCB);
     defwSetConstraintCbk (constraintCB);
     defwSetDefaultCapCbk (defaultCapCB);
     defwSetDesignCbk (designCB);
     defwSetDesignEndCbk (designendCB);
     ...


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

```
    res = defwCloseEncrypt();  // Clean up the encrypted buffer
     ...

    fclose(f);

     return 0;
     }
```

The following example shows the callback routine to mark the end of the DEF design. The type is `defwDesignEndCbkType`.

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


int designendCB (defwCallbackType_e type,
                      defiUserData userData) {
         int    res;


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

Blockages

Blockages routines write a DEF `BLOCKAGES` statement. The `BLOCKAGES` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `BLOCKAGES` statement, see "Blockages" in the *LEF/DEF Language Reference*.

A `BLOCKAGES` statement must start and end with the `defwStartBlockages` and `defwEndBlockages` routines. All blockages must be defined between these routines.

defwStartBlockages

Starts a `BLOCKAGES` statement.

Syntax

```
int defwStartBlockages(
     int count)
```

Arguments

count

Specifies the number of blockages defined in the `BLOCKAGES` statement.

defwEndBlockages

Ends the `BLOCKAGES` statement.

Syntax

```
int defwEndBlockages()
```

defwBlockageDesignRuleWidth

Writes a `DESIGNRULEWIDTH` statement for the blockage. Either a `SPACING` or a `DESIGNRULEWIDTH` statement can be specified for a routing blockage. The `DESIGNRULEWIDTH` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Note: This function will become obsolete in the next parser release. Use `defwBlockagesLayerDesignRuleWidth` instead.

Syntax

```
defwBlockageDesignRuleWidth(
     int effectiveWidth)
```

Arguments

effectiveWidth

Specifies that the blockages have a width of *effectiveWidth* for the purposes of spacing calculations.

defwBlockagesLayerDesignRuleWidth

Writes a DESIGNRULEWIDTH statement for the blockage. Either a SPACING or a DESIGNRULEWIDTH statement can be specified for a routing blockage. The DESIGNRULEWIDTH statement is optional and can be used only once for each routing blockage in the BLOCKAGES statement.

Syntax

```
defwBlockagesLayerDesignRuleWidth(
     int effectiveWidth)
```

Arguments

effectiveWidth

Specifies that the blockages have a width of *effectiveWidth* for the purposes of spacing calculations.

defwBlockageLayer

Writes a LAYER statement that defines a routing blockage. When the *compName* argument is specified, writes a LAYER COMPONENT statement that defines a routing blockage that is associated with a component. Either a LAYER, LAYER COMPONENT, FILLS, SLOTS, or PUSHDOWN statement can be specified for each routing blockage in the BLOCKAGES statement. The LAYER and LAYER COMPONENT statements are optional and each can be used only once for each routing blockage in the BLOCKAGES statement.

Note: This function will become obsolete in the next parser release. Use defwBlockagesLayer and/or defwBlockagesLayerComponent instead.

Syntax

```
int defwBlockageLayer(
     const char* layerName,
     const char* compName)
```

Arguments

layerName

Specifies the layer on which to create the routing blockage.

compName

Optional argument that specifies a component with which to associate the blockage. Specify NULL to ignore this argument.

defwBlockagesLayer

Writes a LAYER statement that defines a routing blockage. Any one of the LAYER, LAYER COMPONENT, FILLS, SLOTS, or PUSHDOWN statements can be specified for each routing blockage in the BLOCKAGES statement. The LAYER statement is optional and can be used only once for each routing blockage in the BLOCKAGES statement.

Syntax

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

Arguments

layerName

Specifies the layer on which to create the routing blockage.

defwBlockagesLayerComponent

Writes a LAYER COMPONENT statement that defines a routing blockage that is associated with a component. Any one of the LAYER, LAYER COMPONENT, FILLS, SLOTS, or PUSHDOWN statements can be specified for each routing blockage in the BLOCKAGES statement. The LAYER COMPONENT statement is optional and can be used only once for each routing blockage in the BLOCKAGES statement.

Syntax

```
int defwBlockagesLayerComponent(
     const char* compName)
```

Arguments

compName

Specifies a component with which to associate the blockage.

defwBlockageLayerExceptpgnet

Writes an `EXCEPTPGNET` statement for a routing blockage on the given layer, which specifies that the blockage only blocks signal net routing and does not block power or ground net routing. Either a `COMPONENT`, `SLOTS`, `FILLS`, `PUSHDOWN`, or `EXCEPTPGNET` statement can be specified for each routing blockage in the `BLOCKAGES` statement. The `EXCEPTPGNET` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Note: This function will become obsolete in the next parser release. Use `defwBlockagesLayerExceptpgnet` instead.

Syntax

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

Arguments

layerName

Specifies the layer on which to create the routing blockage.

defwBlockagesLayerExceptpgnet

Writes an `EXCEPTPGNET` statement for a routing blockage on the given layer, which specifies that the blockage only blocks signal net routing and does not block power or ground net routing. Any one of the `COMPONENT`, `SLOTS`, `FILLS`, `PUSHDOWN`, or `EXCEPTPGNET` statements can be specified for each routing blockage in the `BLOCKAGES` statement. The `EXCEPTPGNET` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Syntax

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

Arguments

layerName

Specifies the layer on which to create the routing blockage.

defwBlockageLayerFills

Writes a `FILLS` statement, which defines a routing blockage on the specified layer where metal fills cannot be placed. Either a `LAYER`, `LAYER COMPONENT`, `FILLS`, `SLOTS`, `PUSHDOWN`, or `EXCEPTPGNET` statement can be specified for each routing blockage in the `BLOCKAGES` statement. The `FILLS` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Note: This function will become obsolete in the next parser release. Use `defwBlockagesLayerFills` instead.

Syntax

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

Arguments

`layerName`
  Specifies the layer on which to create the blockage.

defwBlockagesLayerFills

Writes a `FILLS` statement, which defines a routing blockage where metal fills cannot be placed. Any one of the `LAYER`, `LAYER COMPONENT`, `FILLS`, `SLOTS`, `PUSHDOWN`, or `EXCEPTPGNET` statements can be specified for each routing blockage in the `BLOCKAGES` statement. The `FILLS` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Syntax

```
int defwBlockagesLayerFills()
```

defwBlockageLayerPushdown

Writes a `LAYER PUSHDOWN` statement, which defines the routing blockage as being pushed down into the block from the top level of the design. Either a `LAYER`, `LAYER COMPONENT`, `FILLS`, `SLOTS`, `PUSHDOWN`, or `EXCEPTPGNET` statement can be specified for each routing blockage in the `BLOCKAGES` statement. The `LAYER PUSHDOWN` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Note: This function will become obsolete in the next parser release. Use `defwBlockagesLayerPushdown` instead.

Syntax

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

Arguments

layerName
Specifies the layer on which the blockage lies.

defwBlockagesLayerPushdown

Writes a `LAYER PUSHDOWN` statement, which defines the routing blockage as being pushed down into the block from the top level of the design. Any one of the `LAYER`, `LAYER COMPONENT`, `FILLS`, `SLOTS`, `PUSHDOWN`, or `EXCEPTPGNET` statements can be specified for each routing blockage in the `BLOCKAGES` statement. The `LAYER PUSHDOWN` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Syntax

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

Arguments

layerName
Specifies the layer on which the blockage lies.

defwBlockageLayerSlots

Writes a `SLOTS` statement, which defines a routing blockage where slots cannot be placed. Either a `LAYER`, `LAYER COMPONENT`, `FILLS`, `SLOTS`, `PUSHDOWN`, or `EXCEPTPGNET` statement can be specified for each routing blockage in the `BLOCKAGES` statement. The `SLOTS` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Syntax

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

Arguments

layerName
Specifies the layer on which to create the blockage.

defwBlockagePlacement

Writes a `PLACEMENT` statement, which defines a placement blockage. Either a `PLACEMENT`, `PLACEMENT COMPONENT`, `PLACEMENT PUSHDOWN`, `PLACEMENT PARTIAL`, or `PLACEMENT SOFT` statement can be specified for each placement blockage in the `BLOCKAGES` statement. The `PLACEMENT` statement is optional and can be used only once for each placement blockage in the `BLOCKAGES` statement.

Note: This function will become obsolete in the next parser release. Use `defwBlockagesPlacement` instead.

Syntax

```
defwBlockagePlacement()
```

defwBlockagesPlacement

Writes a `PLACEMENT` statement, which defines a placement blockage. Any one of the `PLACEMENT`, `PLACEMENT COMPONENT`, `PLACEMENT PUSHDOWN`, `PLACEMENT PARTIAL`, or `PLACEMENT SOFT` statements can be specified for each placement blockage in the `BLOCKAGES` statement. The `PLACEMENT` statement is optional and can be used only once for each placement blockage in the `BLOCKAGES` statement.

Syntax

```
defwBlockagesPlacement()
```

defwBlockagePlacementComponent

Writes a `PLACEMENT COMPONENT` statement, which defines a placement blockage associated with a component. Either a `PLACEMENT`, `PLACEMENT COMPONENT`, `PLACEMENT`

PUSHDOWN, PLACEMENT PARTIAL, or PLACEMENT SOFT statement can be specified for each placement blockage in the BLOCKAGES statement. The PLACEMENT COMPONENT statement is optional and can be used only once for each placement blockage in the BLOCKAGES statement.

Note: This function will become obsolete in the next parser release. Use defwBlockagesPlacementComponent instead.

Syntax

```
int defwBlockagePlacementComponent(
     const char* compName)
```

Arguments

compName

Specifies the component with which to associate the blockage.

defwBlockagesPlacementComponent

Writes a PLACEMENT COMPONENT statement, which defines a placement blockage associated with a component. Any one of the PLACEMENT, PLACEMENT COMPONENT, PLACEMENT PUSHDOWN, PLACEMENT PARTIAL, or PLACEMENT SOFT statements can be specified for each placement blockage in the BLOCKAGES statement. The PLACEMENT COMPONENT statement is optional and can be used only once for each placement blockage in the BLOCKAGES statement.

Syntax

```
int defwBlockagesPlacementComponent(
     const char* compName)
```

Arguments

compName

Specifies the component with which to associate the blockage.

defwBlockagePlacementPartial

Writes a `PLACEMENT PARTIAL` statement, which specifies that the initial placement should not use more than *`maxDensity`* percentage of the blockage area for standard cells. Either a `PLACEMENT`, `PLACEMENT PARTIAL`, `PLACEMENT COMPONENT`, `PLACEMENT SOFT`, or `PLACEMENT PUSHDOWN` statement can be specified for each placement blockage. The `PLACEMENT PARTIAL` statement is optional and can be used only once for each placement blockage in the `BLOCKAGES` statement.

Note: This function will become obsolete in the next parser release. Use `defwBlockagesPlacementPartial` instead.

Syntax

```
int defwBlockagePlacementPartial(
     double maxDensity)
```

Arguments

`maxDensity`

Specifies the maximum density value. The initial placement will not use more than *`maxDensity`* percentage of the blockage area for standard cells.
Value: `0.0-100.0`

defwBlockagesPlacementPartial

Writes a `PLACEMENT PARTIAL` statement, which specifies that the initial placement should not use more than *`maxDensity`* percentage of the blockage area for standard cells. Any one of the `PLACEMENT`, `PLACEMENT PARTIAL`, `PLACEMENT COMPONENT`, `PLACEMENT SOFT`, or `PLACEMENT PUSHDOWN` statements can be specified for each placement blockage. The `PLACEMENT PARTIAL` statement is optional and can be used only once for each placement blockage in the `BLOCKAGES` statement.

Syntax

```
int defwBlockagesPlacementPartial(
     double maxDensity)
```

Arguments

maxDensity

Specifies the maximum density value. The initial placement will not use more than *maxDensity* percentage of the blockage area for standard cells.
Value: `0.0-100.0`

defwBlockagePlacementPushdown

Writes a `PLACEMENT PUSHDOWN` statement, which defines the placement blockage as being pushed down into the block from the top level of the design. Either a `PLACEMENT`, `PLACEMENT COMPONENT`, `PLACEMENT PUSHDOWN`, `PLACEMENT PARTIAL`, or `PLACEMENT SOFT` statement can be specified for each placement blockage in the `BLOCKAGES` statement. The `PLACEMENT PUSHDOWN` statement is optional and can be used only once for each placement blockage in a `BLOCKAGES` statement.

Note: This function will become obsolete in the next parser release. Use `defwBlockagesPlacementPushdown` instead.

Syntax

```
int defwBlockagePlacementPushdown()
```

defwBlockagesPlacementPushdown

Writes a `PLACEMENT PUSHDOWN` statement, which defines the placement blockage as being pushed down into the block from the top level of the design. Any one of the `PLACEMENT`, `PLACEMENT COMPONENT`, `PLACEMENT PUSHDOWN`, `PLACEMENT PARTIAL`, or `PLACEMENT SOFT` statement can be specified for each placement blockage in the `BLOCKAGES` statement. The `PLACEMENT PUSHDOWN` statement is optional and can be used only once for each placement blockage in a `BLOCKAGES` statement.

Syntax

```
int defwBlockagesPlacementPushdown()
```

defwBlockagePlacementSoft

Writes a `PLACEMENT  SOFT` statement, which specifies that the initial placement should not use the blockage area, but later timing optimization phases can use the blockage area. Either a `PLACEMENT`, `PLACEMENT PARTIAL`, `PLACEMENT COMPONENT`, `PLACEMENT SOFT`, or

PLACEMENT PUSHDOWN statement can be specified for each placement blockage. The PLACEMENT SOFT statement is optional and can be used only once for each placement blockage in the BLOCKAGES statement.

Note: This function will become obsolete in the next parser release. Use defwBlockagesPlacementSoft instead.

Syntax

```
int defwBlockagePlacementSoft()
```

defwBlockagesPlacementSoft

Writes a PLACEMENT SOFT statement, which specifies that the initial placement should not use the blockage area, but later timing optimization phases can use the blockage area. Any one of the PLACEMENT, PLACEMENT PARTIAL, PLACEMENT COMPONENT, PLACEMENT SOFT, or PLACEMENT PUSHDOWN statements can be specified for each placement blockage. The PLACEMENT SOFT statement is optional and can be used only once for each placement blockage in the BLOCKAGES statement.

Syntax

```
int defwBlockagesPlacementSoft()
```

defwBlockagePolygon

Writes a POLYGON statement. Either a RECT or a POLYGON statement is required with a LAYER, LAYER COMPONENT, FILLS, SLOTS, or PUSHDOWN statement. The POLYGON statement can be used more than once for each routing blockage in the BLOCKAGES statement.

Note: This function will become obsolete in the next parser release. Use defwBlockagesPolygon instead.

Syntax

```
defwBlockagePolygon(
     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, to the y axis, or at a 45-degree angle.

defwBlockagesPolygon

Writes a `POLYGON` statement. Either a `RECT` or a `POLYGON` statement is required with a `LAYER`, `LAYER COMPONENT`, `FILLS`, `SLOTS`, or `PUSHDOWN` statement. The `POLYGON` statement can be used more than once for each routing blockage in the `BLOCKAGES` statement.

Syntax

```
int defwBlockagesPolygon(
     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, to the y axis, or at a 45-degree angle.

defwBlockageRect

Writes a `RECT` statement. Either a `RECT` or a `POLYGON` statement is required with a `LAYER`, `LAYER COMPONENT`, `FILLS`, `SLOTS`, or `LAYER PUSHDOWN` statement. A `RECT` statement is also required with a `PLACEMENT COMPONENT` or `PLACEMENT PUSHDOWN` statement. The `RECT` statement can be used more than once for each blockage in the `BLOCKAGES` statement.

Note: This function will become obsolete in the next parser release. Use `defwBlockagesRect` instead.

Syntax

```
int defwBlockageRect(
     int xl,
     int yl,
     int xh,
     int yh)
```

Arguments

xl yl xh yh

Specifies the absolute coordinates of the blockage geometry.

defwBlockagesRect

Writes a RECT statement. Either a RECT or a POLYGON statement is required with a LAYER, LAYER COMPONENT, FILLS, SLOTS, or LAYER PUSHDOWN statement. A RECT statement is also required with a PLACEMENT COMPONENT or PLACEMENT PUSHDOWN statement. The RECT statement can be used more than once for each blockage in the BLOCKAGES statement.

Syntax

```
int defwBlockagesRect(
     int xl,
     int yl,
     int xh,
     int yh)
```

Arguments

xl yl xh yh

Specifies the absolute coordinates of the blockage geometry.

defwBlockagesLayerMask

Writes the blockage layer color mask.

Syntax

```
int defwBlockagesLayerMask(
     int maskColor)
```

Arguments

maskColor

Specifies the mask color.

defwBlockageSpacing

Writes a `SPACING` statement for the blockage. Either a `SPACING` or a `DESIGNRULEWIDTH` statement can be specified for a routing blockage. The `SPACING` statement is optional and can be used only once for each routing blockage in the `BLOCKAGES` statement.

Syntax

```
defwBlockageSpacing(
     int minSpacing)
```

Arguments

minSpacing

Specifies the minimum spacing between this blockage and any other routing shape.

Bus Bit Characters

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

This routine returns `0` if successful.

defwBusBitChars

Writes a `BUSBITCHARS` statement.

Syntax

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

Arguments

busBitChars

Specifies the pair of characters used to specify bus bits when DEF names are mapped to or from other databases. The characters must be enclosed in double quotation marks. If one of the bus bit characters appears in a DEF name as a regular character, you must use a backslash (\) before the character to prevent the DEF reader from interpreting the character as a bus bit delimiter.

Components

Components routines write a DEF `COMPONENTS` section. The `COMPONENTS` section is optional and can be used only once in a DEF file. For syntax information about the DEF `COMPONENTS` section, see "Components" in the *LEF/DEF Language Reference*.

The `COMPONENTS` section must start and end with the `defwStartComponents` and `defwEndComponents` routines. All components must be defined between these routines.

If the DEF file contains a `REGIONS` statement, the `COMPONENTS` statement must follow it. For more information about the DEF `REGIONS` routines, see "Regions" on page 202.

For examples of the routines described here, see "Components Example" on page 134.

Note: To write a `PROPERTY` statement for the component, you must use one of the property routines between the routines described here. For more information, see "Property Statements" on page 200.

All routines return `0` if successful.

defwStartComponents

Starts the `COMPONENTS` section.

Syntax

```
int defwStartComponents(
     int count)
```

Arguments

count

Specifies the number of components defined in the COMPONENTS section.

defwEndComponents

Ends the COMPONENTS section.

If the *count* specified in defwStartComponents is not the same as the actual number of defwComponent routines used, this routine returns DEFW_BAD_DATA.

Syntax

```
int defwEndComponents(void)
```

defwComponent

Writes a set of statements that define one component. This routine is required and can be used more than once in the COMPONENTS statement.

If you specify 0 for all optional arguments except *weight*, they are ignored. For *weight*, you must specify -1.0.

Syntax

```
int defwComponent(
     const char* name,
     const char* master,
     const char* eeq,
     const char* source,
     const char* status,
     int statusX,
     int statusY,
     int statusOrient,
     double weight,
     const char* region,)
```

Arguments

eeq

Optional argument that specifies that the component being defined should be electrically equivalent to *eeq* (a previously defined component). Specify `NULL` to ignore this argument.

master

Specifies the name of a model defined in the library.

name

Specifies the component name, which is an instance of *master*.

region

Optional argument that specifies the name of a previously defined region in which the component must lie. Specify `NULL` to ignore this argument.

status

Optional argument that specifies the component state. Specify `NULL` to ignore this argument.
Value: Specify one of the following:

`COVER`	Specifies that the component has a location and is a part of the cover macro. It cannot be moved by automatic tools or interactive commands.
`FIXED`	Specifies that the component has a location and cannot be moved by automatic tools, but can me moved using interactive commands.
`PLACED`	Specifies that the component has a location, but can be moved using automatic layout tools.
`UNPLACED`	Specifies that the component does not have a location.

statusOrient

Optional argument that specifies the orientation of the component. Specify `-1` to ignore this argument.
Value: `0` to `7`. For more information, see "Orientation Codes" on page 19.

statusX statusY

Optional arguments that specify the location of the component. Specify `0` to ignore these arguments.

source

Optional argument that specifies the source of the component. Specify `NULL` to ignore this argument.
Value: Specify one of the following:

`DIST`	Component is a physical component (that is, it only connects to power or ground nets), such as filler cells, well-taps, and decoupling caps.
`NETLIST`	Component is specified in the original netlist. This is the default value, and is normally not written out in the DEF file.
`TIMING`	Component is a logical rather than physical change to the netlist, and is typically used as a buffer for a clock-tree, or to improve timing on long nets.
`USER`	Component is generated by the user for some user-defined reason.

weight

Optional argument that specifies the weight of the component, which determines if automatic placement attempts to keep the component near the specified location. *weight* is only meaningful when the component is placed. All non-zero weights have the same effect during automatic placement. Specify `0` to ignore this argument.

defwComponentStr

Also writes a set of statements that define one component. This routine is the same as the `defwComponent` routine, with the exception of the *foreignOrients* argument, which takes a string instead of an integer.This routine is required and can be used more than once in the `COMPONENTS` statement.

If you specify `0` for all optional arguments except *weight*, they are ignored. For weight, you must specify `-1.0`.

Syntax

```
int defwComponent(
     const char* name,
     const char* master,
     const char* eeq,
     const char* source,
     const char* status,
     int statusX,
     int statusY,
     const char* statusOrient,
     double weight,
     const char* region,)
```

Arguments

eeq

Optional argument that specifies that the component being defined should be electrically equivalent to *eeq* (a previously defined component). Specify NULL to ignore this argument.

master

Specifies the name of a model defined in the library.

name

Specifies the component name, which is an instance of *master*.

region

Optional argument that specifies the name of a previously defined region in which the component must lie. Specify NULL to ignore this argument.

status

Optional argument that specifies the component state. Specify NULL to ignore this argument.
Value: Specify one of the following:

COVER	Specifies that the component has a location and is a part of the cover macro. It cannot be moved by automatic tools or interactive commands.
FIXED	Specifies that the component has a location and cannot be moved by automatic tools, but can me moved using interactive commands.

`PLACED`	Specifies that the component has a location, but can be moved using automatic layout tools.
`UNPLACED`	Specifies that the component does not have a location.

`statusOrient`

Optional argument that specifies the orientation of the component. Specify `NULL` to ignore this argument.
Value: `N`, `W`, `S`, `E`, `FN`, `FW`, `FS`, or `FE`

`statusX statusY`

Optional arguments that specify the location of the component. Specify `0` to ignore these arguments.

`source`

Optional argument that specifies the source of the component. Specify `NULL` to ignore this argument.
Value: Specify one of the following:

`DIST`	Component is a physical component (that is, it only connects to power or ground nets), such as filler cells, well-taps, and decoupling caps.
`NETLIST`	Component is specified in the original netlist. This is the default value, and is normally not written out in the DEF file.
`TIMING`	Component is a logical rather than physical change to the netlist, and is typically used as a buffer for a clock-tree, or to improve timing on long nets.
`USER`	Component is generated by the user for some user-defined reason.

`weight`

Optional argument that specifies the weight of the component, which determines if automatic placement attempts to keep the component near the specified location. *`weight`* is only meaningful when the component is placed. All non-zero weights have the same effect during automatic placement. Specify `0` to ignore this argument.

defwComponentHalo

Writes a `HALO` statement for a component. The `HALO` statement creates a placement blockage around the component. The `HALO` statement is optional and can be used only once for each component in the `COMPONENT` statement. If you call this routine, you cannot call `defwComponentHaloSoft`.

Syntax

```
defwComponentHalo(
     int left,
     int bottom,
     int right,
     int top)
```

Arguments

left bottom right top

Specifies the amount the halo extends from the left, bottom, right, and top edges of the LEF macro.

defwComponentHaloSoft

Writes a `HALO SOFT` statement. This routine is similar to `defwComponentHalo`, except that it also writes the `SOFT` option. The `HALO SOFT` statement is optional and can be used only once for each component. If you call this routine, you cannot call `defwComponentHalo`.

Syntax

```
int defwComponentHaloSoft(
     int left,
     int bottom,
     int right,
     int top)
```

Arguments

left bottom right top

Specifies the amount the halo extends from the left, bottom, right, and top edges of the LEF macro.

defwComponentRouteHalo

Writes a `ROUTEHALO` statement. The `ROUTEHALO` statement is optional and can be used only once for each component.

Syntax

```
int defwComponentRouteHalo(
     int haloDist,
     const char* minLayer,
     const char* maxLayer)
```

Arguments

haloDist

Specifies the halo distance, as an integer in DEF database units.

minLayer

Specifies the minimum layer. The routing halo exists for the routing layers between *minLayer* and *maxLayer*. *minLayer* must be a lower routing layer than *maxLayer*. *minLayer* must be a string that matches a LEF routing layer name.

maxLayer

Specifies the maximum layer. The routing halo exists for the routing layers between *minLayer* and *maxLayer*. *maxLayer* must be a string that matches a LEF routing layer name.

Components Example

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

```
int componentCB (defwCallbackType_e type,
                      defiUserData userData) {

          int   res;
          const char** foreigns;
          int   *foreignX, *foreignY, *foreignOrient;


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

```
}
foreigns = (const char**)malloc(sizeof(char*)*1);
foreignX = (int*)malloc(sizeof(int)*1);
foreignY = (int*)malloc(sizeof(int)*1);
foreignOrient = (int*)malloc(sizeof(int)*1);
res = defwStartComponents(2);
CHECK_RES(res);
res = defwComponent("Z38A01", "DFF3", 0, NULL, NULL, NULL,
                    NULL, NULL, 0, NULL, NULL, NULL, NULL,
                    "PLACED", 18592, 5400, 6, 0, NULL, 0, 0, 0,
                    0);
CHECK_RES(res);
foreigns[0] = strdup("gds2name");
foreignX[0] = -500;
foreignY[0] = -500;
foreignOrient[0] = 3;
res = defwComponent("cell3", "CHM6A", 0, NULL, NULL, NULL,
                    NULL, "TIMING", 1, foreigns, foreignX,
                    foreignY, foreignOrient, "PLACED", 240, 10,
                    0, 0, "region1", 0, 0, 0, 0);
CHECK_RES(res);
res = defwStringProperty("cc", "This is the copy list");
CHECK_RES(res);
res = defwIntProperty("index", 9);
CHECK_RES(res);
res = defwRealProperty("size", 7.8);
CHECK_RES(res);
res = defwEndComponents();
CHECK_RES(res);
free((char*)foreigns[0]);
free((char*)foreigns);
free((char*)foreignX);
free((char*)foreignY);
free((char*)foreignOrient);
return 0;}
```

Design Name

The Design routine writes a DEF `DESIGN` statement. The `DESIGN` statement is required and can be used only once in a DEF file. For syntax information about the `DESIGN` statement, see “Design” in the *LEF/DEF Language Reference*.

This routine returns `0` if successful.

defwDesignName

Writes a `DESIGN` statement.

Syntax

```
int defwDesignName(
     const char* name)
```

Arguments

name

Specifies a name for the design.

Die Area

Die Area routines write a DEF `DIEAREA` statement. The `DIEAREA` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `DIEAREA` statement, see "Die Area" in the *LEF/DEF Language Reference*.

If the DEF file contains a `PROPERTYDEFINITIONS` statement, the `DIEAREA` statement must follow it. For more information about the DEF `PROPERTYDEFINITIONS` statement, see "Property Definitions" on page 196.

This routine returns `0` if successful.

defwDieArea

Writes a `DIEAREA` statement.

Syntax

```
int defwDieArea (
     int xl,
     int yl,
     int xh,
     int yh )
```

Arguments

xl, yl, xh, yh

Specifies the points of two corners of the bounding rectangle for the design. Geometric shapes (such as blockages, pins, and special net routing) can be outside of the die area, to allow proper modeling of pushed down routing from top-level designs into sub blocks. However, routing tracks should still be inside the die area.

defwDieAreaList

Writes a `DIEAREA` statement that includes more than two points.

Syntax

```
defwDieAreaList(
     int num_points,
     int* xl,
     int*yh)
```

Arguments

num_points

Specifies the number of points specified.

xl yh

Specifies the points of a polygon that forms the die area. Geometric shapes (such as blockages, pins, and special net routing) can be outside of the die area, to allow proper modeling of pushed down routing from top-level designs into sub blocks. However, routing tracks should still be inside the die area.

Die Area Example

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

```
int dieareaCB (defwCallbackType_e type,
                    defiUserData userData) {
         int    res;

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

```
}
res = defwDieArea(-190000, -120000, 190000, 70000);
CHECK_RES(res);
return 0;}
```

Divider Character

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

This routine returns `0` if successful.

defwDividerChar

Writes a `DIVIDERCHAR` statement.

Syntax

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

Arguments

dividerChar

Specifies the character used to express hierarchy when DEF names are mapped to or from other databases. The character must be enclosed in double quotation marks.
If the divider character appears in a DEF name as a regular character, you must use a backslash (\) before the character to prevent the DEF reader from interpreting the character as a hierarchy delimiter.

Extensions

The Extension routines write a series of statements that define the `EXTENSIONS` statement in the DEF file. The `EXTENSIONS` statement is optional and can be used only once in a DEF file. For syntax information about the `EXTENSIONS` statement, see "Extensions" in the *LEF/DEF Language Reference*.

You must use the `defwStartBeginext` and `defwEndBeginext` routines to create an `EXTENSIONS` statement. You must define all extensions between these routines.

For examples of the routines described here, see "Extensions Example" on page 141.

All routines return `0` if successful.

defwStartBeginext

Starts the `EXTENSIONS` statement.

Syntax

```
int defwStartBeginext(
     const char* name)
```

Arguments

name
Specifies the extension name.

defwEndBeginext

Ends the `BEGINEXT` statement.

Syntax

```
int defwEndBeginext()
```

defwBeginextCreator

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

Syntax

```
int defwBeginextCreator(
     const char* creatorName)
```

Arguments

creatorName

Specifies a string value that defines the creator value.

defwBeginextDate

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 defwBeginextDate()
```

defwBeginextRevision

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

Syntax

```
int defwBeginextRevision(
     int vers1,
     int vers2)
```

Arguments

vers1, vers2

Specifies the values used for the revision number string.

defwBeginextSyntax

Adds customized syntax to the DEF 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 Example

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

```
int extensionCB (defwCallbackType_e type,
                     defiUserData userData) {
         int    res;

         // Check if the type is correct
         if (type != defwExtCbkType) {
             printf("Type is not defwExtCbkType, terminate
               writing.\n");
             return 1;
         }
         res = defwStartBeginext("tag");
         CHECK_RES(res);
         res = defwBeginextCreator("CADENCE");
         CHECK_RES(res);
         res = defwBeginextDate();
         CHECK_RES(res);
         res = defwBeginextSyntax("OTTER", "furry");
         CHECK_RES(res);
         res = defwStringProperty("arrg", "later");
         CHECK_RES(res);
         res = defwBeginextSyntax("SEAL", "cousin to WALRUS");
         CHECK_RES(res);
         res = defwEndBeginext();
         CHECK_RES(res);
         return 0;}
```

Fills

Fills routines write a DEF `FILLS` statement. The `FILLS` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `FILLS` statement, see "Fills" in the *LEF/DEF Language Reference*.

The DEF `FILLS` statement must start and end with the `defwStartFills` and `defwEndFills` routines. All fills must be defined between these routines.

All routines return `0` if successful.

defwStartFills

Starts a `FILLS` statement.

Syntax

```
int defwStartFills(
     int count)
```

Arguments

count

Specifies the number of fills defined in the `FILLS` statement.

defwEndFills

Ends the `FILLS` statement.

Syntax

```
int defwEndFills()
```

defwFillLayer

Writes a `LAYER` statement. The `LAYER` statement is required for each fill and can be used more than once in a `FILLS` statement.

Syntax

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

Arguments

layerName Specifies the layer on which to create the fill.

defwFillLayerOPC

Writes an `OPC` keyword for a `FILLS LAYER` statement, which specifies that FILL shapes require OPC correction during mask generation. `defwFillLayer` must be called before this routine. This routine is optional and can be called only once after the `defwFillLayer` or `defwFillVia` routine.

Syntax

```
int defwFillLayerOPC()
```

defwFillPoints

Specifies the points for a `FILLS VIA` statement. This routine is required after `defwFillVia` and can be called more than once.

Syntax

```
int defwFillPoints(
     int num_points,
     double* x1,
     double* y1)
```

Arguments

num_points

Specifies the number of points provided.

x1 y1

Specify the placement locations (x y points) for the via.

defwFillPolygon

Writes a `POLYGON` statement. Either a `POLYGON` or a `RECT` statement is required with a `LAYER` statement. The `POLYGON` statement is required and can be used more than once for each fill in the `FILLS` statement.

Syntax

```
defwFillPolygon(
     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.

defwFillRect

Writes a `RECT` statement. Either a `POLYGON` or a `RECT` statement is required with a LAYER statement. The `RECT` statement is required and can be used more than once for each fill in the `FILLS` statement.

Syntax

```
int defwFillRect(
     int xl,
     int yl,
     int xh,
     int yh)
```

Arguments

xl, yl, xh, yh

Specifies the coordinates of the fill.

defwFillVia

Writes a `FILLS VIA` statement. The `FILLS VIA` statement is optional and can be used more than once. Call `defwFillPoints` after this routine.

Syntax

```
int defwFillVia(
     const char* viaName)
```

Arguments

viaName

The name of the via, which must be previously defined in the DEF VIA or LEF VIA section.

defwFillViaOPC

Writes the `OPC` keyword for a `FILLS VIA` statement, which specifies that FILL shapes require OPC correction during mask generation. This routine is optional and can only be called after `defwFillVia`.

Syntax

```
int defwFillViaOPC()
```

GCell Grid

The Gcell Grid routine writes a DEF `GCELLGRID` statement. The `GCELLGRID` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `GCELLGRID` statement, see GCell Grid in the *LEF/DEF Language Reference*.

If the DEF file contains a `TRACKS` statement, the `GCELLGRID` statement must follow it. For more information about the DEF `TRACKS` statement, see “Tracks” on page 242.

This routine returns `0` if successful.

defwGcellGrid

Writes a `GCELLGRID` statement.

Syntax

```
int defwGcellGrid(
     const char* master,
     int doStart,
     int doCount,
     int doStep)
```

Arguments

doCount

Specifies the number of columns or rows in the grid.

doStart

Specifies the starting location of the grid (that is, the first column or row).

doStep

Specifies the step spacing between the grid units.

master

Specifies the direction of the tracks for the global router grid that overlays the array.
Value: Specify one of the following:

X	Specifies a vertical grid.
Y	Specifies a horizontal grid.

Gcell Grid Example

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

```
int gcellgridCB (defwCallbackType_e type,
                     defiUserData userData) {
         int    res;

         // Check if the type is correct
         if (type != defwGcellGridCbkType) {
             printf("Type is not defwGcellGridCbkType, terminate
               writing.\n");
             return 1;
         }
         res = defwGcellGrid("X", 0, 100, 600);
         CHECK_RES(res);
         return 0;}
```

Groups

The Groups routines write a DEF GROUPS statement. The GROUPS statement is optional and can be used only once in a DEF file. For syntax information about the DEF GROUPS statement, see Groups in the *LEF/DEF Language Reference*.

You must begin and end a DEF GROUPS statement with the defwStartGroups and defwEndGroups routines. You must define all groups between these routines.

For examples of the routines described here, see "Groups Example" on page 149.

Note: To write a PROPERTY statement for the component, you must use one of the property routines immediately following the defwGroup* routines that define the group. For more information, see "Property Statements" on page 200.

All routines return 0 if successful.

defwStartGroups

Starts the GROUPS statement.

Syntax

```
int defwStartGroups(
     int count)
```

Arguments

count

Specifies the number of groups defined in the GROUPS statement.

defwEndGroups

Ends the GROUPS statement.

Syntax

```
int defwEndGroups()
```

defwGroup

Writes a series of statements that define the specified group. This routine is required and can be used more than once in a `GROUPS` statement.

Syntax

```
int defwGroup(
     const char* groupName,
     int numExpr,
     const char** groupExpr)
```

Arguments

groupExpr

Specifies a component name, a list of component names, or a regular expression for a set of components.

groupName

Specifies the name for a group of components.

numExpr

Specifies the number of components in the group.

defwGroupRegion

Writes a `REGION` statement for the group defined. This statement is optional and can be used only once per group name.

Syntax

```
int defwGroupRegion(
     int xl,
     int yl,
     int xh,
     int yh,
     const char* regionName)
```

Arguments

regionName

Specifies the name of a previously defined region in which the group must lie.

xl xh yl yh

Specifies the coordinates of a rectangular region in which the group must lie. Specify the coordinates or *regionName*; do not specify both.

Groups Example

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

```
int dividerCB (defwCallbackType_e type,
                    defiUserData userData) {
         int   res;
         const char **groupExpr;

         // Check if the type is correct
         if (type != defwGroupCbkType) {
             printf("Type is not defwGroupCbkType, terminate
               writing.\n");
             return 1;
         }
         groupExpr = (const char**)malloc(sizeof(char*)*2);
         res = defwStartGroups(2);
         CHECK_RES(res);
         groupExpr[0] = strdup("cell2");
         groupExpr[1] = strdup("cell3");
         res = defwGroup("group1", 2, groupExpr);
         CHECK_RES(res);
         free((char*)groupExpr[0]);
         free((char*)groupExpr[1]);
         res = defwGroupRegion(0, 0, 0, 0, "region1");
         CHECK_RES(res);
         res = defwStringProperty("ggrp", "xx");
         CHECK_RES(res);
         res = defwIntProperty("side", 2);
         CHECK_RES(res);
         res = defwRealProperty("maxarea", 5.6);
         CHECK_RES(res);
         groupExpr[0] = strdup("cell1");
         res = defwGroup("group2", 1, groupExpr);
         CHECK_RES(res);
         free((char*)groupExpr[0]);
         res = defwGroupRegion(0, 10, 1000, 1010, NULL);
         CHECK_RES(res);
         res = defwGroupSoft("MAXHALFPERIMETER", 4000, "MAXX", 10000,
```

```
 NULL, NULL);
CHECK_RES(res);
res = defwEndGroups();
CHECK_RES(res);
free((char*)groupExpr);
// Write a new line
res = defwNewLine();
CHECK_RES(res);
return 0;}
```

History

The History routine writes a DEF `HISTORY` statement. The `HISTORY` statement is optional and can be used more than once in a DEF file. For syntax information about the DEF `HISTORY` statement, see History in the *LEF/DEF Language Reference*.

This routine returns `0` if successful.

defwHistory

Writes a `HISTORY` statement.

Syntax

```
int defwHistory(
     const char* string)
```

Arguments

string

Lists a historical record about the design. Each line indicates one historical record. Any text excluding a semicolon (;) can be included. `Linefeed` and `Return` do not terminate the statement.

History Example

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

```
int historyCB (defwCallbackType_e type,
                    defiUserData userData) {
        int    res;
```

```
// Check if the type is correct
if (type != defwHistoryCbkType) {
    printf("Type is not defwHistoryCbkType, terminate
      writing.\n");
    return 1;
}
res = defwHistory("DEF version 5.3");
CHECK_RES(res);
return 0;}
```

Nets

Nets routines write a DEF `NETS` statement. The `NETS` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `NETS` statement, see "Nets" in the *LEF/DEF Language Reference*.

A `NETS` statement must start and end with the `defwStartNets` and `defwEndNets` routines. All nets must be defined between these routines. Each individual net must start and end with either `defwNet` or `defwNetMustjoinConnection`, and `defwNetEndOneNet`.

For examples of the routines described here, see "Nets Example" on page 161.

In addition to the routines in this section, you can also include routines that form a *regularWiring* statement, a `SUBNET` statement, and a `PROPERTY` statement. For information about these routines, see "Regular Wiring" on page 164, "Subnet" on page 169, and "Property Statements" on page 200.

All routines return `0` if successful.

defwStartNets

Starts a `NETS` statement. A `NET` statement must start and end with `defwStartNets` and `defwEndNets`.

Syntax

```
int defwStartNets(
     int count)
```

Arguments

count

Specifies the number of nets defined in the `NETS` statement.

defwEndNets

Ends the `NETS` statement. A `NET` statement must start and end with `defwStartNets` and `defwEndNets`.

Syntax

```
int defwEndNets()
```

defwNet

Starts a net description in the `NETS` statement. Each net description must start with either `defwNet` or `defwNetMustJoinConnection`, and end with `defwNetEndOneNet`.

If you specify this routine, you can optionally specify the following routine:

- defwNetConnection on page 153

Syntax

```
int defwNet(
     const char* netName)
```

Arguments

netName

Specifies the name of the net.

defwNetMustjoinConnection

Writes a `MUSTJOIN` statement in the `NETS` statement. Each net description must start with either `defwNet` or `defwNetMustJoinConnection`, and end with `defwNetEndOneNet`.

Syntax

```
int defwNetMustjoinConnection(
     const char* compName,
     const char* pinName)
```

Arguments

compName, *pinName*

Identifies the net as a mustjoin by specifying one of its pins, using a component name and pin name.

defwNetEndOneNet

Ends a net description in the `NETS` statement. Each net description must start with either `defwNet` or `defwNetMustJoinConnection`, and end with `defwNetEndOneNet`.

Syntax

```
int defwNetEndOneNet()
```

defwNetConnection

Defines the net specified in `defwNet`. This routine can be used more than once for each net in a `NETS` statement.

Syntax

```
int defwNetConnection(
     const char* compName,
     const char* pinName,
     int synthesized)
```

Arguments

compName

Specifies the name of a regular component pin on the net. If you omit this value, the DEF writer writes the `PIN` statement.

pinName

Specifies the name of an I/O pin on the net.

synthesized

Optional argument that marks the pin as part of a synthesized scan chain.
Value: Specify one of the following:

0	Argument is ignored.
1	Writes a SYNTHESIZED statement.

defwNetEstCap

Writes an ESTCAP statement. The ESTCAP statement is optional and can be used only once for each net in the NETS statement.

Syntax

```
int defwNetEstCap(
     double wireCap)
```

Arguments

wireCap

Specifies the estimated wire capacitance for the net. ESTCAP can be loaded with simulation data to generate net constraints for timing-driven layout.

defwNetFixedBump

Writes a FIXEDBUMP statement that indicates a bump cannot be assigned to a different pin. The FIXEDBUMP statement is optional and can be used only once for a net.

Syntax

```
int defwNetFixedBump()
```

defwNetFrequency

Writes a FREQUENCY statement. The FREQUENCY statement is optional and can be used only once for a net.

Syntax

```
int defwNetFrequency(
     double frequency)
```

Arguments

frequency

Specifies the frequency of the net, in hertz. The frequency value is used by the router to choose the correct number of via cuts required for a given net, and by validation tools to verify that the AC current density rules are met.

defwNetNondefaultRule

Writes a `NONDEFAULTRULE` statement. The `NONDEFAULTRULE` statement is optional and can be used only once for a net.

Syntax

```
int defwNetNondefaultRule(
     const char* ruleName)
```

Arguments

ruleName

Specifies that the net and wiring are created according to the specified nondefault rule defined in LEF.

defwNetOriginal

Writes an `ORIGINAL` statement. The `ORIGINAL` statement is optional and can be used only once for a net.

Syntax

```
int defwNetOriginal(
     const char* netName)
```

Arguments

netName

Specifies the name of the original net partitioned to create multiple nets, including the net being defined.

defwNetPattern

Writes a `PATTERN` statement. The `PATTERN` statement is optional and can be used only once for a net.

Syntax

```
int defwNetPattern(
     const char* name)
```

Arguments

name

Specifies the routing pattern used for the net.
Value: Specify one of the following:

`BALANCED`	Used to minimize skews in timing delays for clock nets.
`STEINER`	Used to minimize net length.
`TRUNK`	Used to minimize delay for global nets.
`WIREDLOGIC`	Used in ECL designs to connect output and mustjoin pins before routing to the remaining pins.

defwNetSource

Writes a `SOURCE` statement. The `SOURCE` statement is optional and can be used only once for a net.

Syntax

```
int defwNetSource(
     const char* name)
```

Arguments

name

Specifies the source of the net.
Value: Specify one of the following:

DIST	Net is the result of adding physical components (that is, components that only connect to power or ground nets), such as filler cells, well-taps, tie-high and tie-low cells, and decoupling caps.
NETLIST	Net is defined in the original netlist. This is the default value, and is not normally written out in the DEF file.
TEST	Net is part of a scanchain.
TIMING	Net represents a logical rather than physical change to netlist, and is used typically as a buffer for a clock-tree, or to improve timing on long nets.
USER	Net is user defined.

defwNetUse

Writes a `USE` statement. The `USE` statement is optional and can be used only once for a net.

Syntax

```
int defwNetUse(
     const char* name)
```

Arguments

name

Specifies how the net is used.
Value: Specify one of the following:

ANALOG	Used as a analog signal net.
CLOCK	Used as a clock net.

`GROUND`	Used as a ground net.
`POWER`	Used as a power net.
`RESET`	Used as a reset net.
`SCAN`	Used as a scan net.
`SIGNAL`	Used as digital signal net.
`TIEOFF`	Used as a tie-high or tie-low net.

defwNetVpin

Writes a `VPIN` statement. The `VPIN` statement is optional and can be used more than once for a net.

Syntax

```
int defwNetVpin(
     const char* vpinName,
     const char* layerName,
     int layerXl,
     int layerYl,
     int layerXh,
     int layerYh,
     const char* status,
     int statusX,
     int statusY,
     int orient)
```

Arguments

layerName

Optional argument that specifies the layer on which the virtual pin lies. Specify `NULL` to ignore this argument.

layerXl layerYl layerXh layerYh

Specifies the physical geometry of the virtual pin.

orient

Optional argument that specifies the orientation of the virtual pin. Specify `-1` to ignore this argument.
Value: `0` to `7`. For more information, see "Orientation Codes" on page 19.

status

Optional argument that specifies the placement status of the virtual pin. Specify `NULL` to ignore this argument.
Value: specify one of the following:

`COVER`	Specifies that the pin has a location and is a part of the cover macro. It cannot be moved by automatic tools or interactive commands.
`FIXED`	Specifies that the pin has a location and cannot be moved by automatic tools but can be moved by interactive commands.
`PLACED`	Specifies that the pin has a location, but can be moved during automatic layout.

statusX statusY

Optional arguments that specify the placement location of the virtual pin. If you specify *status*, you must specify these arguments. Specify `0` to ignore these arguments.

vpinName

Specifies the name of the virtual pin to define.

defwNetVpinStr

Also writes a `VPIN` statement. This routine is the same as the `defwNetVpin` routine, with the exception of the *orient* argument, which takes a string instead of an integer. The `VPIN` statement is optional and can be used more than once for a net.

Syntax

```
int defwNetVpin(
     const char* vpinName,
     const char* layerName,
     int layerXl,
     int layerYl,
     int layerXh,
     int layerYh,
     const char* status,
     int statusX,
     int statusY,
     const char* orient)
```

Arguments

layerName

Optional argument that specifies the layer on which the virtual pin lies. Specify `NULL` to ignore this argument.

layerXl layerYl layerXh layerYh

Specifies the physical geometry of the virtual pin.

orient

Optional argument that specifies the orientation of the virtual pin. Specify `NULL` to ignore this argument.
Value: `N`, `W`, `S`, `E`, `FN`, `FW`, `FS`, or `FE`

status

Optional argument that specifies the placement status of the virtual pin. Specify `NULL` to ignore this argument.
Value: specify one of the following:

`COVER`	Specifies that the pin has a location and is a part of the cover macro. It cannot be moved by automatic tools or interactive commands.
`FIXED`	Specifies that the pin has a location and cannot be moved by automatic tools but can be moved by interactive commands.
`PLACED`	Specifies that the pin has a location, but can be moved during automatic layout.

statusX statusY

Optional arguments that specify the placement location of the virtual pin. If you specify *status*, you must specify these arguments. Specify `0` to ignore these arguments.

vpinName

Specifies the name of the virtual pin to define.

defwNetWeight

Writes a `WEIGHT` statement. The `WEIGHT` statement is optional and can be used only once for a net.

Syntax

```
int defwNetWeight(
     double weight)
```

Arguments

weight

Specifies the weight of the net. Automatic layout tools attempt to shorten the lengths of nets with high weights. A value of `0` indicates that the net length for that net can be ignored. A value of `1` specifies that the net should be treated normally. A larger weight specifies that the tool should try harder to minimize the net length of that net.
For normal use, timing constraints are generally a better method to use for controlling net length than net weights. For the best results, you should typically limit the maximum weight to 10, and not add weights to more than 3 percent of the nets.

defwNetXtalk

Writes a `XTALK` statement. The `XTALK` statement is optional and can be used only once for a net.

Syntax

```
int defwNetXtalk(
     int num)
```

Arguments

num

Specifies the crosstalk class number for the net. If you specify the default value (`0`), the `XTALK` statement will not be written to the DEF file.
Value: `0` to `200`

Nets Example

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

```
int netCB (defwCallbackType_e type,
                defiUserData userData) {
        int   res;
        const char **coorX, **coorY;
```

```
const char **coorValue;

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

res = defwStartNets(3);
CHECK_RES(res);

coorX = (const char**)malloc(sizeof(char*)*5);
coorY = (const char**)malloc(sizeof(char*)*5);
coorValue = (const char**)malloc(sizeof(char*)*5);
res = defwNet("my_net");
CHECK_RES(res);
res = defwNetConnection("I1", "A", 0);
CHECK_RES(res);
res = defwNetConnection("BUF", "Z", 0);
CHECK_RES(res);
res = defwNetNondefaultRule("RULE1");
CHECK_RES(res);
res = defwNetShieldnet("VSS");
CHECK_RES(res);
res = defwNetPathStart("ROUTED");
CHECK_RES(res);
...
 = defwNetNoshieldStart("M2");
CHECK_RES(res);
coorX[0] = strdup("14100");
coorY[0] = strdup("341440");
coorX[1] = strdup("14000");
coorY[1] = strdup("*");
res = defwNetNoshieldPoint(2, coorX, coorY);
CHECK_RES(res);
res = defwNetNoshieldEnd();
CHECK_RES(res);
res = defwNetEndOneNet();
CHECK_RES(res);

res = defwNet("MUSTJOIN");
CHECK_RES(res);
res = defwNetConnection("cell4", "PA1", 0);
CHECK_RES(res);
res = defwNetEndOneNet();
CHECK_RES(res);

res = defwNet("XX100");
CHECK_RES(res);
res = defwNetConnection("Z38A05", "G", 0);
```

```
CHECK_RES(res);
res = defwNetConnection("Z38A03", "G", 0);
CHECK_RES(res);
res = defwNetConnection("Z38A01", "G", 0);
CHECK_RES(res);
res = defwNetVpin("V_SUB3_XX100", NULL, -333, -333, 333,
                333, "PLACED", 189560, 27300, 0);
CHECK_RES(res);
res = defwNetSubnetStart("SUB1_XX100");
CHECK_RES(res);
...
 // An example for Regular Wiring can be found in the
 // Regular Wiring section.

res = defwNetPathEnd();
CHECK_RES(res);
res = defwNetNoshieldStart("M2");
CHECK_RES(res);
coorX[0] = strdup("14100");
coorY[0] = strdup("341440");
coorX[1] = strdup("14000");
coorY[1] = strdup("*");
res = defwNetNoshieldPoint(2, coorX, coorY);
CHECK_RES(res);
res = defwNetNoshieldEnd();
CHECK_RES(res);
res = defwNetEndOneNet();
CHECK_RES(res);

res = defwNet("MUSTJOIN");
CHECK_RES(res);
res = defwNetConnection("cell4", "PA1", 0);
CHECK_RES(res);
res = defwNetEndOneNet();
CHECK_RES(res);

res = defwNet("XX100");
CHECK_RES(res);
res = defwNetConnection("Z38A05", "G", 0);
CHECK_RES(res);
res = defwNetConnection("Z38A03", "G", 0);
CHECK_RES(res);
res = defwNetConnection("Z38A01", "G", 0);
CHECK_RES(res);
res = defwNetVpin("V_SUB3_XX100", NULL, -333, -333, 333,
           333, "PLACED", 189560, 27300, 0);
CHECK_RES(res);
res = defwNetSubnetStart("SUB1_XX100");
CHECK_RES(res);
...
 // An example for Subnet can be found in the Subnet section
```

```
CHECK_RES(res);
res = defwNetSubnetEnd();
CHECK_RES(res);
res = defwEndNets();
CHECK_RES(res);
return 0;}
```

Regular Wiring

Routines described in this section form a *regularWiring* statement that can be used to define regular wiring for a net or subnet. The *regularWiring* statement is optional and can be used more than once in a `NETS` statement. For syntax information about the DEF `NETS` statement, see "Nets" in the *LEF/DEF Language Reference*.

A *regularWiring* statement must start and end with the `defwNetPathStart` and `defwNetPathEnd` routines. All regular wiring must be defined between these routines.

For examples of the routines described here, see "Regular Wiring Example" on page 169.

The regular wiring routines can be included between the following pairs of routines:

- `defwNet` and `defwEndOneNet`
- `defwNetMustjoinConnection` and `defwEndOneNet`
- `defwNetSubnetStart` and `defwSubnetEnd`

All routines return `0` if successful.

defwNetPathStart

Starts a *regularWiring* statement.

Syntax

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

Arguments

type

Specifies the regular wiring type.
Value: Specify one of the following:

COVER	Specifies that the wiring cannot be moved by either automatic layout or interactive commands.
FIXED	Specifies that the wiring cannot be moved by automatic layout, but can be changed by interactive commands.
ROUTED	Specifies that the wiring can be moved by the automatic layout tools.
NOSHIELD	Specifies that the last wide segment of the net is not shielded.

defwNetPathEnd

Ends the *regularWiring* statement.

Syntax

```
int defwNetPathEnd()
```

defwNetPathLayer

Writes a `LAYER` statement. The `LAYER` statement is required and can be used more than once in the *regularWiring* statement.

Syntax

```
int defwNetPathLayer(
     const char* layerName,
     int isTaper,
     const char* rulename)
```

Arguments

layerName

Specifies the layer name on which the wire lies.

isTaper

Optional argument that writes the keyword TAPER, which specifies that the next contiguous wire segment is created using the default rule.
Value: Specify one of the following:

0	Ignores the argument.
1	Writes the keyword TAPER. If you specify 1, you must specify NULL for the *rulename* argument.

ruleName

Optional argument that specifies that the next contiguous wire segment is created using the specified nondefault rule (*ruleName*). Specify NULL to ignore this argument. If you specify a *rulename*, you must specify 0 for the *isTaper* argument.

defwNetPathPoint

Defines the center line coordinates of the route on the layer specified with defwNetPathLayer. This routine is required and can be used only once for each layer in the *regularWiring* statement.

Syntax

```
int defwNetPathPoint(
     int numPts,
     const char** pointX,
     const char** pointY,
     const char** value)
```

Arguments

numPts

Specifies the number of points in the wire path (route)

pointX pointY

Specifies the coordinates of the path points.

value

Optional argument that specifies the amount by which the wire is extended past the end point of the segment. This value must be greater than or equal to 0 (zero). Specify NULL to ignore this argument.

defwNetPathStyle

Writes a STYLE statement for the layer specified with defwNetPathLayer. The STYLE statement is optional and can be used only once for each layer in the *regularWiring* statement.

Syntax

```
defwNetPathStyle(
     int styleNum)
```

Arguments

styleNum

Specifies a previously defined style from the STYLES section in this DEF file. If a style is specified, the wire's shape is defined by the center line coordinates and the style.

defwNetPathVia

Specifies a via to place at the last point on the layer specified with defwNetPathLayer. This routine is optional and can be used only once for each layer in the *regularWiring* statement.

Syntax

```
int defwNetPathVia(
     const char* viaName)
```

Arguments

viaName

Specifies the via to place at the last specified path coordinate.

defwNetPathViaWithOrient

Specifies the orientation of the via specified with `defwNetPathVia`. This routine is optional and can be used only once for each via in the *`regularWiring`* statement.

Syntax

```
defwNetPathViaWithOrient(
     const char* name,
     int orient)
```

Arguments

`name`

Specifies the via.

`orient`

Specifies the orientation.
Value: `0` to `7`. For more information, see "Orientation Codes" on page 19

defwNetPathViaWithOrientStr

Also specifies the orientation of the via specified with `defwNetPathVia`. This routine is the same as the `defwNetPathViaWithOrient` routine, with the exception of the *`orient`* argument, which takes a string instead of an integer. The `defwNetPathViaWithOrientStr` is optional and can be used only once for each via in the *`regularWiring`* statement.

Syntax

```
defwNetPathViaWithOrient(
     const char* name,
     int orient)
```

Arguments

`name`

Specifies the via.

`orient`

Specifies the orientation. Specify `NULL` to ignore this argument.
Value: `N`, `W`, `S`, `E`, `FN`, `FW`, `FS`, or `FE`

Regular Wiring Example

The following example only shows the usage of some functions related to regular wiring in a net. This example is part of the net callback routine.

```
int netCB (defwCallbackType_e type,
                defiUserData userData) {
         int   res;
         const char **coorX, **coorY;
         const char **coorValue;

         ...
         res = defwNetPathStart("NEW");
         CHECK_RES(res);
         res = defwNetPathLayer("M1", 1, NULL);
         CHECK_RES(res);
         coorX[0] = strdup("2400");
         coorY[0] = strdup("282400");
         coorValue[0] = NULL;
         coorX[1] = strdup("240");
         coorY[1] = strdup("*");
         coorValue[1] = NULL;
         res = defwNetPathPoint(2, coorX, coorY, coorValue);
         CHECK_RES(res);
         free((char*)coorX[0]);
         free((char*)coorY[0]);
         free((char*)coorX[1]);
         free((char*)coorY[1]);
         res = defwNetPathEnd();
         CHECK_RES(res);
         ...

         return 0;}
```

Subnet

The Subnet routines write a `SUBNET` statement which further defines a net. A `SUBNET` statement is optional and can be used more than once in a `NETS` statement. For information about the DEF `NETS` statement, see "Nets" in the *LEF/DEF Language Reference*.

You must begin and end a `SUBNET` statement with the `defwNetSubnetStart` and `defwSubnetEnd` routines. You must define all subnets between these routines.

For examples of the routines described here, see "Subnet Example" on page 171.

In addition to the routines described in this section, you can include a NONDEFAULTRULE statement and a *regularWiring* statement within a SUBNET statement. For more information about these routines, see defwNetNondefaultRule on page 155, or "Regular Wiring" on page 164.

All routines return 0 if successful.

defwNetSubnetStart

Starts a SUBNET statement. This statement is optional and can be used only once in a NETS statement.

Syntax

```
int defwNetSubnetStart(
     const char* name)
```

Arguments

name

Specifies the name of the subnet.

defwNetSubnetEnd

Ends a SUBNET statement.

Syntax

```
int defwNetSubnetEnd()
```

defwNetSubnetPin

Specifies a component for the SUBNET statement. This routine is optional and can be used more than once in a SUBNET statement.

Syntax

```
int defwNetSubnetPin(
     const char* component,
     const char* name)
```

Arguments

component

Specifies either a component name, or the value `PIN` or `VPIN`.

name

Specifies either a pin name if *component* is set to `PIN`, or a virtual pin name if *component* is set to `VPIN`.

Subnet Example

The following example only shows the usage of some functions related to subnet in a net. This example is part of the net callback routine.

```
int netCB (defwCallbackType_e type,
                defiUserData userData) {
        int   res;
        const char **coorX, **coorY;
        const char **coorValue;

        ...
        res = defwNetSubnetStart("SUB1_XX100");
        CHECK_RES(res);
        res = defwNetSubnetPin("Z38A05", "G");
        CHECK_RES(res);
        res = defwNetSubnetPin("VPIN", "V_SUB1_XX100");
        CHECK_RES(res);
        res = defwNetPathStart("ROUTED");
        CHECK_RES(res);
        res = defwNetPathLayer("M1", 0, "RULE1");
        CHECK_RES(res);
        coorX[0] = strdup("54040");
        coorY[0] = strdup("30300");
        coorValue[0] = strdup("0");
        coorX[1] = strdup("*");
        coorY[1] = strdup("30900");
        coorValue[1] = NULL;
        res = defwNetPathPoint(2, coorX, coorY, coorValue);
        CHECK_RES(res);
        free((char*)coorX[0]);
        free((char*)coorY[0]);
```

```
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
res = defwNetPathVia("nd1VIA12");
CHECK_RES(res);
...
res = defwNetPathEnd();
CHECK_RES(res);
res = defwNetSubnetEnd();
...

return 0;}
```

Nondefault Rules

Nondefault rule routines write a DEF `NONDEFAULTRULES` statement. The `NONDEFAULTRULES` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `NONDEFAULTRULES` statement, see "Nondefault Rules" in the *LEF/DEF Language Reference*.

The `NONDEFAULTRULES` statement must start and end with the `defwStartNonDefaultRules` and `defwEndNonDefaultRules` routines. All nondefault rules must be defined between these two routines. Each individual nondefault rule must start with `defwNonDefaultRule`.

Note: To write a `PROPERTY` statement for the nondefault rule, you must use one of the property routines immediately following the `defwNonDefaultRule` routine. For more information, see "Property Statements" on page 200.

All routines return `0` if successful.

defwStartNonDefaultRules

Starts a `NONDEFAULTRULES` statement.

Syntax

```
defwStartNonDefaultRules(
     int count)
```

Arguments

count

Specifies the number of rules defined in the NONDEFAULTRULES statement.

defwEndNonDefaultRules

Ends the NONDEFAULTRULES statement.

Syntax

```
defwEndNonDefaultRules()
```

defwNonDefaultRule

Starts a nondefault rule definition. This routine is required for each nondefault rule and can be used more than once in the NONDEFAULTRULES statement.

Syntax

```
defwNonDefaultRule(
     const char* ruleName,
     int hardSpacing)
```

Arguments

ruleName

Specifies the name for this nondefault rule. This name can be used in the NETS section wherever a nondefault rule name is allowed. The reserved name DEFAULT can be used to indicate the default routing rule used in the NETS section.

hardSpacing

Optional argument that specifies that any spacing values that exceed the LEF LAYER ROUTING spacing requirements are “hard” rules instead of “soft” rules. Specify 0 to ignore this argument.

defwNonDefaultRuleLayer

Writes a LAYER statement for the nondefault rule. The LAYER statement is required and can be used more than once for each nondefault rule in the NONDEFAULTRULES statement.

Syntax

```
defwNonDefaultRuleLayer(
     const char* layerName,
     double width,
     double diagWidth,
     double spacing,
     double wireExt)
```

Arguments

layerName

‘Specifies the layer for the various width and spacing values. *layerName* must be a routing layer.

width

Specifies the required minimum width allowed for *layerName*.

diagWidth

Optional argument that specifies the diagonal width for *layerName*, when 45-degree routing is used. Specify 0 to ignore this argument.

spacing

Optional argument that specifies the minimum spacing for *layerName*. The LEF LAYER SPACING or SPACINGTABLE definitions always apply; therefore it is only necessary to add a SPACING value if the desired spacing is larger than the LAYER rules already require. Specify 0 to ignore this argument.

wireExt

Optional argument that specifies the distance by which wires are extended at vias on *layerName*. Specify 0 to ignore this argument.

defwNonDefaultRuleMinCuts

Writes a MINCUTS statement. The MINCUTS statement is optional and can be used more than once for each nondefault rule in the NONDEFAULTRULES statement.

Syntax

```
defwNonDefaultRuleMinCuts(
     const char* cutLayerName,
     int numCuts)
```

Arguments

cutLayerName

Specifies the cut layer.

numCuts

Specifies the minimum number of cuts allowed for any via using *cutLayerName*. All vias (generated or fixed vias) used for this nondefault rule must have at least *numCuts* cuts in the via.

defwNonDefaultRuleVia

Writes a VIA statement for the nondefault rule. The VIA statement is optional and can be used more than once for each nondefault rule in the NONDEFAULTRULES statement.

Syntax

```
defwNonDefaultRuleVia(
     const char* viaName)
```

Arguments

viaName

Specifies a previously defined LEF or DEF via to use with this rule.

defwNonDefaultRuleViaRule

Writes a VIARULE statement. The VIARULE statement is optional and can be used more than once for each nondefault rule in the NONDEFAULTRULES statement.

Syntax

```
defwNonDefaultRuleViaRule(
     const char* viaRuleName)
```

Arguments

viaRuleName

Specifies a previously defined LEF VIARULE GENERATE to use with this routing rule. If no via or via rule is specified for a given routing-cut-routing layer combination, then a

VIARULE GENERATE DEFAULT via rule must exist for that combination, and it is implicitly inherited.

Pins

Pin routines write a DEF PINS statement. The PINS statement is optional and can be used only once in a DEF file. For syntax information about the DEF PINS statement, see "Pins" in the *LEF/DEF Language Reference*.

A PINS statement must start and end with the defwStartPins and defwEndPins routines. All pins must be defined between these routines. Each individual pin must start with a defwPin routine.

If the DEF file contains a COMPONENTS statement, the PINS statement must follow it. For more information about DEF COMPONENTS routines, see "Components" on page 127.

For examples of the routines described here, see "Pins Example" on page 193.

Note: To write a PROPERTY statement for the pin, you must use one of the property routines immediately following the defwPin routine. For more information, see "Property Statements" on page 200.

All routines return 0 if successful.

defwStartPins

Starts a PINS statement.

Syntax

```
int defwStartPins(
     int count)
```

Arguments

count

Specifies the number of pins defined in the PINS statement.

defwEndPins

Ends the PINS statement. If *count* is not the same as the actual number of defwPin routines used, defwEndPins returns DEFW_BAD_DATA.

Syntax

```
int defwEndPins(void)
```

defwPin

Starts a pin description in the PINS statement. Each pin description must start with defwPin. This routine is required and can be used more than once in a PINS statement.

Syntax

```
int defwPin(
     const char* pinName,
     const char* netName,
     int special,
     const char* direction,
     const char* use,
     const char* status,
     int statusX,
     int statusY,
     int orient)
```

Arguments

direction

Optional argument that specifies the pin type. Specify NULL to ignore this argument.
Value: Specify one of the following:

FEEDTHRU	Pin that goes completely across the cell.
INPUT	Pin that accepts signals coming into the cell.
INOUT	Pin that drives signals out of the cell.
OUTPUT	Pin that can accept signals going either in or out of the cell.

netName

Specifies the corresponding internal net name.

orient

Optional argument that specifies the orientation for the pin. Specify `-1` to ignore this argument.
Value: `0` to `7`. For more information, see "Orientation Codes" on page 19.

pinName

Specifies the name for the external pin.

special

Optional argument that identifies the pin as a special pin. Specify `0` to ignore this argument.
Value: Specify one of the following: I

`0`	Argument is ignored.
`1`	Writes a `SPECIAL` statement.

status

Optional argument that specifies the placement status of the pin. Specify `NULL` to ignore this argument.
Value: Specify one of the following:

`COVER`	Specifies that the pin has location and is a part of a cover macro. It cannot be moved by automatic layout tools or by interactive commands.
`FIXED`	Specifies that the pin has a location and cannot be moved by automatic tools, but can be moved by interactive commands.
`PLACED`	Specifies that the pin has a location, but can be moved during automatic layout.

statusX statusY

Optional arguments that specify the placement location of the pin. If you specify *status*, you must specify these arguments.Specify `0` to ignore these arguments.

use

Optional argument that specifies how the pin is used. Specify `NULL` to ignore this argument.
Value: Specify one of the following:

`ANALOG`	Pin is used for analog connectivity.
`CLOCK`	Pin is used for clock net connectivity.
`GROUND`	Pin is used for connectivity to the chip-level ground distribution network.
`POWER`	Pin is used for connectivity to the chip-level power distribution network.
`RESET`	Pin is used as reset pin.
`SCAN`	Pin is used as scan pin.
`SIGNAL`	Pin is used for regular net connectivity.
`TIEOFF`	Pin is used as tie-high or tie-low pin.

defwPinStr

Also starts a pin description in the `PINS` statement. This routine is the same as the `defwPin` routine, with the exception of the *`orient`* argument, which takes a string instead of an integer. Each pin description must start with `defwPin`. This routine is required and can be used more than once in a `PINS` statement.

Syntax

```
int defwPin(
     const char* pinName,
     const char* netName,
     int special,
     const char* direction,
     const char* use,
     const char* status,
     int statusX,
     int statusY,
     const char* orient)
```

Arguments

direction

Optional argument that specifies the pin type. Specify `NULL` to ignore this argument.
Value: Specify one of the following:

`FEEDTHRU`	Pin that goes completely across the cell.
`INPUT`	Pin that accepts signals coming into the cell.
`INOUT`	Pin that drives signals out of the cell.
`OUTPUT`	Pin that can accept signals going either in or out of the cell.

netName

Specifies the corresponding internal net name.

orient

Optional argument that specifies the orientation for the pin. Specify `NULL` to ignore this argument.
Value: `N`, `W`, `S`, `E`, `FN`, `FW`, `FS`, or `FE`

pinName

Specifies the name for the external pin.

special

Optional argument that identifies the pin as a special pin. Specify `0` to ignore this argument.
Value: Specify one of the following: I

`0`	Argument is ignored.
`1`	Writes a `SPECIAL` statement.

`status`

Optional argument that specifies the placement status of the pin. Specify `NULL` to ignore this argument.
Value: Specify one of the following:

`COVER`	Specifies that the pin has location and is a part of a cover macro. It cannot be moved by automatic layout tools or by interactive commands.
`FIXED`	Specifies that the pin has a location and cannot be moved by automatic tools, but can be moved by interactive commands.
`PLACED`	Specifies that the pin has a location, but can be moved during automatic layout.

`statusX statusY`

Optional arguments that specify the placement location of the pin. If you specify *`status`*, you must specify these arguments.Specify `0` to ignore these arguments.

`use`

Optional argument that specifies how the pin is used. Specify `NULL` to ignore this argument.
Value: Specify one of the following:

`ANALOG`	Pin is used for analog connectivity.
`CLOCK`	Pin is used for clock net connectivity.
`GROUND`	Pin is used for connectivity to the chip-level ground distribution network.
`POWER`	Pin is used for connectivity to the chip-level power distribution network.
`RESET`	Pin is used as reset pin.
`SCAN`	Pin is used as scan pin.
`SIGNAL`	Pin is used for regular net connectivity.
`TIEOFF`	Pin is used as tie-high or tie-low pin.

defwPinAntennaModel

Writes an `ANTENNAMODEL` statement. The `ANTENNAMODEL` statement is optional and can be used more than once in a pin definition.

Syntax

```
int defwPinAntennaModel(
     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: `OXIDE3` and `OXIDE4` are currently not supported. If you specify either of these models, the tool parses and ignores it.

defwPinAntennaPinDiffArea

Writes an `ANTENNAPINDIFFAREA` statement. The `ANTENNAPINDIFFAREA` statement is optional and can be used more than once in a `PIN` section.

Syntax

```
int defwPinAntennaPinDiffArea(
     int value,
     const char* layerName)
```

Argument

value

Specifies the diffusion (diode) area to which the pin is connected on a layer.

layerName

Optional argument that specifies the layer. Specify `NULL` to ignore this argument.

defwPinAntennaPinGateArea

Writes an `ANTENNAPINGATEAREA` statement. The `ANTENNAPINGATEAREA` statement is optional, and can be used once after each `defwPinAntennaModel` routine in a `PINS` section.

Syntax

```
int defwPinAntennaPinGateArea(
     int value,
     const char* layerName)
```

Arguments

value

Specifies the gate area to which the pin is connected on a layer.

layerName

Optional argument that specifies the layer. Specify `NULL` to ignore this argument.

defwPinAntennaPinMaxAreaCar

Writes an `ANTENNAPINMAXAREACAR` statement. The `ANTENNAPINMAXAREACAR` statement is optional, and can be used once after each `defwPinAntennaModel` routine in a `PINS` section.

Syntax

```
int defwPinAntennaPinMaxAreaCar(
     int value,
     const char* layerName)
```

Arguments

value

Specifies the maximum cumulative antenna ratio, using the metal area below the current pin layer.

layerName

Specifies the pin layer.

defwPinAntennaPinMaxCutCar

Writes an `ANTENNAPINMAXCUTCAR` statement. The `ANTENNAPINMAXCUTCAR` statement is optional, and can be used once after each `defwPinAntennaModel` routine in a `PINS` section.

Syntax

```
int defwPinAntennaPinMaxCutCar(
     int value,
     const char* layerName)
```

Arguments

value

Specifies the maximum cumulative antenna ratio, using the cut area below the current pin layer.

layerName

Specifies the pin layer.

defwPinAntennaPinMaxSideAreaCar

Writes an `ANTENNAPINMAXSIDEAREACAR` statement. The `ANTENNAPINMAXSIDEAREACAR` statement is optional, and can be used once after each `defwPinAntennaModel` routine in a `PINS` section.

Syntax

```
int defwPinAntennaPinMaxSideAreaCar(
     int value,
     const char* layerName)
```

Arguments

value

Specifies the maximum cumulative antenna ratio, using the metal side wall area below the current pin layer.

layerName

Specifies the pin layer.

defwPinAntennaPinPartialCutArea

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

Syntax

```
int defwPinAntennaPinPartialCutArea(
     int 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. Specify NULL to ignore this argument.

defwPinAntennaPinPartialMetalArea

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

Syntax

```
int defwPinAntennaPinPartialMetalArea(
     int value,
     const char* layerName)
```

Arguments

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. Specify NULL to ignore this argument.

defwPinAntennaPinPartialMetalSideArea

Writes an ANTENNAPINPARTIALMETALSIDEAREA statement. The ANTENNAPINPARTIALMETALSIDEAREA statement is optional and can be used more than once for each pin in a PINS statement.

Syntax

```
int defwPinAntennaPinPartialMetalSideArea(
     int value,
     const char* layerName)
```

Arguments

value

Specifies the partial metal side wall 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. Specify NULL to ignore this argument.

defwPinGroundSensitivity

Writes a GROUNDSENSITIVITY statement for a pin in the PINS statement. The GROUNDSENSITIVITY statement is optional and can be used only once for each pin in the PINS statement.

Syntax

```
defwPinGroundSensitivity(
     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.

defwPinLayer

Writes a `LAYER` statement for a pin in the PINS statement. Either a `LAYER` or a `POLYGON` statement can be specified for a pin. The `LAYER` statement is optional and can be used more than once for each pin in the `PINS` statement.

Syntax

```
defwPinLayer(
     const char* layerName,
     int spacing,
     int designRuleWidth,
     int xl,
     int yl,
     int xh,
     int yh)
```

Arguments

layerName

Specifies the routing layer used for the pin.

spacing

Optional argument that specifies the minimum spacing allowed between this pin and any other routing shape. If you specify a minimum spacing, you must specify `0` for *designRuleWidth*. Specify `0` to ignore this argument.

designRuleWidth

Optional argument that specifies that this pin has a width of designRuleWidth for the purpose of spacing calculations. If you specify a *designRuleWidth* value, you must specify `0` for *spacing*. Specify `0` to ignore this argument.

xl yl xh yh

Specifies the physical geometry for the pin on the specified layer.

defwPinNetExpr

Writes a `NETEXPR` statement for a pin in the `PINS` statement. The `NETEXPR` statement is optional and can be used only once for each pin in the `PINS` statement.

Syntax

```
defwPinNetExpr(
     const char* pinExpr)
```

Arguments

pinExpr

Specifies a net expression property name (such as power1 or power2). If *pinExpr* matches a net expression property higher up in the netlist (for example, in Verilog, VHDL, or OpenAccess), then the property is evaluated, and the software identifies a net to which to connect this pin.

defwPinPolygon

Writes a `POLYGON` statement for a pin in the `PINS` statement. Either a `LAYER` or a `POLYGON` statement can be specified for a pin. The `POLYGON` statement is optional and can be used more than once for each pin in the `PINS` statement.

Syntax

```
defwPinPolygon(
     const char* layerName,
     int spacing,
     int designRuleWidth,
     int num_polys,
     double* xl,
     double* yl)
```

Arguments

layerName

Specifies the layer on which to generate a polygon.

spacing

Optional argument that specifies the minimum spacing allowed between this pin and any other routing shape. If you specify a minimum spacing, you must specify `0` for *designRuleWidth*. Specify `0` to ignore this argument.

designRuleWidth

Optional argument that specifies that this pin has a width of designRuleWidth for the purpose of spacing calculations. If you specify a *designRuleWidth* value, you must specify `0` for *spacing*. Specify `0` to ignore this argument.

num_polys

Specifies the number of polygon sides.

xl yl

Specifies a sequence of points to generate a polygon for the pin. The polygon edges must be parallel to the x axis, the y axis, or at a 45-degree angle.

defwPinPort

Writes a `PORT` statement for a pin in the `PINS` statement. The `PORT` statement is optional and can be used more than once in a `PINS` statement.

Syntax

```
int defwPinPort()
```

defwPinPortLayer

Writes a `LAYER` statement for a `PINS PORT` statement. Either a `LAYER, POLYGON`, or `VIA` statement can be specified for a pin port. This routine is optional and is called after `defwPinPort`.

Syntax

```
int defwPinPortLayer(
     const char* layerName,
     int spacing,
     int designRuleWidth,
     int xl,
     int yl,
     int xh,
     int yh)
```

Arguments

layerName

Specifies the layer name.

spacing

Optional argument that specifies the minimum spacing allowed between this pin port and any other routing shape. If you specify *spacing*, you must specify `0` for *designRuleWidth*. Specify `0` to ignore this argument.

designRuleWidth

Optional argument that specifies that this pin port has a width of *designRuleWidth* for the purpose of spacing calculations. If you specify *designRuleWidth*, you must specify `0` for *spacing*. Specify `0` to ignore this argument.

xl yl xh yh

Specifies the physical geometry for the pin port on the specified layer.

defwPinPortLocation

Writes a `FIXED`, `PLACED`, or `COVER` statement for a `PINS PORT` statement. This routine is optional and is called after `defwPinPort`.

Syntax

```
int defwPinPortLocation(
     const char* status,
     int statusX,
     int statusY,
     const char* orient)
```

Arguments

status

Specifies the placement status of the pin.
Value: specify one of the following:

`COVER`	Specifies that the pin has a location and is a part of the cover macro. It cannot be moved by automatic tools or interactive commands.

`FIXED`	Specifies that the pin has a location and cannot be moved by automatic tools but can be moved by interactive commands.
`PLACED`	Specifies that the pin has a location, but can be moved during automatic layout.

statusX statusY

Specifies the placement location of the pin. If you specify *status*, you must specify these arguments.

orient

Specifies the orientation of the pin.
Value: `0` to `7`. For more information, see "Orientation Codes" on page 19.

defwPinPortPolygon

Writes a `POLYGON` statement for a `PINS PORT` statement. Either a `LAYER, POLYGON`, or `VIA` statement can be specified for a pin port. This routine is optional and is called after `defwPinPort`.

Syntax

```
int defwPinPortPolygon(
     const char* layerName,
     int spacing,
     int designRuleWidth,
     int num_polys,
     double* xl,
     double* yl)
```

Arguments

layerName

Specifies the layer name.

spacing

Optional argument that specifies the minimum spacing allowed between this pin port and any other routing shape. If you specify a minimum spacing, you must specify `0` for *designRuleWidth*. Specify `0` to ignore this argument.

designRuleWidth

Optional argument that specifies that this pin port has a width of *designRuleWidth* for the purpose of spacing calculations. If you specify *designRuleWidth*, you must specify `0` for *spacing*. Specify `0` to ignore this argument.

num_polys

Specifies the number of polygon sides.

x1 y1

Specifies a sequence of points to generate a polygon for the pin port. The polygon edges must be parallel to the x axis, the y axis, or at a 45-degree angle.

defwPinPortVia

Writes a `VIA` statement for a `PINS PORT` statement. Either a `LAYER, POLYGON`, or `VIA` statement can be specified for a pin port. This routine is optional and is called after `defwPinPort`.

Syntax

```
int defwPinPortVia(
     const char* viaName,
     int x1,
     int y1)
```

Arguments

viaName

Specifies the via name. The via name must have been defined in the associated LEF files or this DEF file before this function is called.

x1 y1

Specifies the point at which the via is to be placed.

defwPinSupplySensitivity

Writes a `SUPPLYSENSITIVITY` statement for a pin in the `PINS` statement. The `SUPPLYSENSITIVITY` statement is optional and can be used only once for each pin in the `PINS` statement.

Syntax

```
defwPinSupplySensitivity(
     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.

defwPinVia

Writes a `VIA` statement for a pin in the `PINS` statement. The `VIA` statement is optional and can be used more than once for a pin.

Syntax

```
int defwPinVia(
     const char* viaName,
     int x1,
     int y1)
```

Arguments

viaName

Specifies the via name. The via name must have been defined in the associated LEF files or this DEF file before this function is called.

x1 y1

Specifies the point at which the via is to be placed.

Pins Example

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

```
int pinCB (defwCallbackType_e type,
                defiUserData userData) {
         int    res;

         // Check if the type is correct
         if (type != defwPinCbkType) {
```

```
        printf("Type is not defwPinCbkType, terminate
           writing.\n");
        return 1;
    }

     res = defwStartPins(1);
    CHECK_RES(res);
    res = defwPin("scanpin", "SCAN", 0, "INPUT", NULL, NULL, 0,
                    0, -1, NULL, 0, 0, 0, 0);
    CHECK_RES(res);
    res = defwEndPins();
    CHECK_RES(res);
    return 0;}
```

Pin Properties

The Pin Properties routines write a DEF `PINPROPERTIES` statement. The `PINPROPERTIES` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `PINPROPERTIES` statement, see "Pin Properties" in the *LEF/DEF Language Reference*.

You must begin and end a DEF `PINPROPERTIES` statement with the `defwStartPinProperties` and `defwEndPinProperties` routines. You must define all pin properties between these routines. Each property definition must start with a `defwPinProperty` routine.

If the DEF file contains a `PINS` statement, the `PINPROPERTIES` statement must follow it. For more information about the DEF `PINS` writer routines, see "Pins" on page 176.

For examples of the routines described here, see"Pin Properties Example" on page 196.

Note: To write a `PROPERTY` statement for a pin, you must use one of the property routines immediately following the `defwPinProperty` routine, which specifies the pin name. For more information, see "Property Statements" on page 200.

All routines return `0` if successful.

defwStartPinProperties

Starts a `PINPROPERTIES` statement.

Syntax

```
int defwStartPinProperties(
     int count)
```

Arguments

count
Specifies the number of pin properties defined in the PINPROPERTIES statement.

defwEndPinProperties

Ends the PINPROPERTIES statement. If *count* specified in defwStartPinProperties is not the same as the actual number of defwPinProperty routines used, defwEndPinProperties returns DEFW_BAD_DATA. This routine does not require any arguments.

Syntax

```
int defwEndPinProperties(void)
```

defwPinProperty

Begins a property definition. This routine is required and can be used more than once in a PINPROPERTIES statement.

Syntax

```
int defwPinProperty(
     const char* component,
     const char* pinName)
```

Arguments

component
Specifies either the string to use for the component pin name, or the keyword PIN.

pinName
Specifies the I/O pin name. Specify this value only when *component* is set to PIN.

Pin Properties Example

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

```
int pinpropCB (defwCallbackType_e type,
                    defiUserData userData) {
         int    res;

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

         res = defwStartPinProperties(2);
         CHECK_RES(res);
         res = defwPinProperty("cell1", "PB1");
         CHECK_RES(res);
         res = defwStringProperty("dpBit", "1");
         CHECK_RES(res);
         res = defwRealProperty("realProperty", 3.4);
         CHECK_RES(res);
         res = defwPinProperty("cell2", "vdd");
         CHECK_RES(res);
         res = defwIntProperty("dpIgnoreTerm", 2);
         CHECK_RES(res);
         res = defwEndPinProperties();
         CHECK_RES(res);
         return 0;}
```

Property Definitions

The Property Definitions routines write a DEF `PROPERTYDEFINITIONS` statement. The `PROPERTYDEFINITIONS` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `PROPERTYDEFINITIONS` statement, see Property Definitions in the *LEF/DEF Language Reference*.

You must begin and end a DEF `PROPERTYDEFINITIONS` statement with the `defwStartPropDef` and `defwEndPropDef` routines. You must define all properties between these routines.

If the DEF file contains a `HISTORY` statement, the `PROPERTYDEFINITIONS` statement must follow it. For more information about the DEF `HISTORY` routine, see “History” on page 150.

For examples of the routines described here, see “Property Definitions Example” on page 199.

All routines return `0` if successful.

defwStartPropDef

Starts a `PROPERTYDEFINITIONS` statement. This routine does not require any arguments.

Syntax

```
int defwStartPropDef(void)
```

defwEndPropDef

Ends the `PROPERTYDEFINITIONS` statement. This routine does not require any arguments.

Syntax

```
int defwEndPropDef(void)
```

defwIntPropDef

Writes an integer property definition. This routine is optional and can be used more than once in a `PROPERTYDEFINITIONS` statement.

Syntax

```
int defwIntPropDef(
     const char* objType,
     const char* propName,
     double leftRange,
     double rightRange,
     const char* value)
```

Arguments

objType

Specifies the type of object for which you can define properties.
Value: `DESIGN`, `COMPONENT`, `NET`, `SPECIALNET`, `GROUP`, `ROW`, `COMPONENTPIN`, `NONDEFAULTRULE`, or `REGION`

propName

Specifies a unique property name for the object type.

leftRange rightRange

Optional arguments that limit integer property values to a specified range. That is, the value must be greater than or equal to *leftRange* and less than or equal to *rightRange*. Specify 0 to ignore these arguments.

value

Optional argument that specifies a numeric value for an object. Specify NULL to ignore this argument.

defwRealPropDef

Writes a real property definition. This routine is optional and can be used more than once in a PROPERTYDEFINITIONS statement.

Syntax

```
int defwRealPropDef(
     const char* objType,
     const char* propName,
     double leftRange,
     double rightRange,
     const char* value)
```

Arguments

objType

Specifies the type of object for which you can define properties.
Value: Specify DESIGN, COMPONENT, NET, SPECIALNET, GROUP, ROW, COMPONENTPIN, NONDEFAULTRULE, or REGION

propName

Specifies a unique property name for the object type.

leftRange rightRange

Optional arguments that limit real number property values to a specified range. That is, the value must be greater than or equal to *leftRange* and less than or equal to *rightRange*. Specify 0 to ignore these arguments.

value

Optional argument that specifies a numeric value for an object. Specify `NULL` to ignore this argument.

defwStringPropDef

Writes a string property definition. This routine is optional and can be used more than once in a `PROPERTYDEFINITIONS` statement.

Syntax

```
int defwStringPropDef(
     const char* objType,
     const char* propName,
     double leftRange,
     double rightRange,
     const char* value)
```

Arguments

objType

Specifies the type of object for which you can define properties.
Value: `DESIGN`, `COMPONENT`, `NET`, `SPECIALNET`, `GROUP`, `ROW`, `COMPONENTPIN`, `NONDEFAULTRULE`, or `REGION`

propName

Specifies a unique property name for the object type.

leftRange rightRange

Optional arguments that limit string property values to a specified range. That is, the value must be greater than or equal to *leftRange* and less than or equal to *rightRange*. Specify `0` to ignore these arguments.

value

Optional argument that specifies a character value for an object. Specify `NULL` to ignore this argument.

Property Definitions Example

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

```
int pinCB (defwCallbackType_e type,
                defiUserData userData) {
         int    res;

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

         res = defwStartPropDef();
         check_res(res);
         defwAddComment("defwPropDef is broken into 3 routines,
          defwStringPropDef");
         defwAddComment("defwIntPropDef, and defwRealPropDef");
         res = defwStringPropDef("REGION", "scum", 0, 0, NULL);
         CHECK_RES(res);
         res = defwIntPropDef("REGION", "center", 0, 0, NULL);
         CHECK_RES(res);
         res = defwRealPropDef("REGION", "area", 0, 0, NULL);
         CHECK_RES(res);
         res = defwStringPropDef("GROUP", "ggrp", 0, 0, NULL);
         CHECK_RES(res);
         res = defwEndPropDef();
         CHECK_RES(res);
         return 0;}
```

Property Statements

The Property Statements routines write `PROPERTY` statements when used after the `defwRow`, `defwRegion`, `defwComponent`, `defwPin`, `defwPinProperty`, `defwSpecialNet`, `defwNet`, `defwNonDefaultRule`, or `defwGroup` routines.

For examples of the routines described here, see "Property Statements Example" on page 202.

defwIntProperty

Writes a `PROPERTY` statement with an integer value. This statement is optional and can be used more than once.

Syntax

```
int defwIntProperty(
     const char* propName,
     int propValue)
```

Arguments

propName

Specifies a unique property name for the object.

propValue

Specifies an integer value for the object.

defwRealProperty

Writes a `PROPERTY` statement with a real number value*.* This statement is optional and can be used more than once.

Syntax

```
int defwRealProperty(
     const char* propName,
     double propValue)
```

Arguments

propName

Specifies a unique property name for the object.

propValue

Specifies a real value for the object.

defwStringProperty

Writes a `PROPERTY` statement with a string value. This statement is optional and can be used more than once.

Syntax

```
int defwStringProperty(
     const char* propName,
     const char* propValue)
```

propName

Specifies a unique property name for the object.

propValue

Specifies a string value for the object.

Property Statements Example

The following example shows how to create a property inside a Rows callback routine.

```
int rowCB (defwCallbackType_e type,
                defiUserData userData) {
         int    res;

         ...
         res = defwRealProperty("minlength", 50.5);
         CHECK_RES(res);
         res = defwStringProperty("firstName", "Only");
         CHECK_RES(res);
         res = defwIntProperty("idx", 1);
         CHECK_RES(res);
         ...

         return 0;}
```

Regions

The Regions routines write a DEF `REGIONS` statement. The `REGIONS` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `REGIONS` statement, see “Regions” in the *LEF/DEF Language Reference*.

You must begin and end a DEF `REGIONS` statement with the `defwStartRegions` and `defwEndRegions` routines. You must define all regions between these routines. Each region definition must start with a `defwRegions` routine.

If the DEF file contains a `VIAS` statement, the `REGIONS` statement must follow it. For more information about the DEF `VIAS` routines, see “Vias” on page 245.

For examples of the routines described here, see “Regions Example” on page 205.

Note: To write a PROPERTY statement for the region, you must use one of the property routines immediately following the defwRegion routines. For more information, see "Property Statements" on page 200.

All routines return 0 if successful.

defwStartRegions

Starts a REGIONS statement.

Syntax

```
int defwStartRegions(
     int count)
```

Arguments

count

Specifies the number of regions defined in the REGIONS statement.

defwEndRegions

Ends the REGIONS statement. If *count* specified in defwStartRegions is not the same as the actual number of defwRegionName routines used, this routine returns DEFW_BAD_DATA. This routine does not require any arguments.

Syntax

```
int defwEndRegions(void)
```

defwRegionName

Starts a region description. This routine must be called the number of times specified in the defwStartRegions *count* argument.

Syntax

```
int defwRegionName(
     const char* regionName)
```

Arguments

regionName

Specifies the name of the region.

defwRegionPoints

Specifies the set of points bounding the region. This routine is required and can be used more than once to define a region.

Syntax

```
int defwRegionPoints(
     int xl,
     int yl,
     int xh,
     int yh)
```

Arguments

xl yl xh yh

Specifies the corner points of the region.

defwRegionType

Writes a `TYPE` statement. The `TYPE` statement is optional and can be used only once per region.

Syntax

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

Arguments

type

Specifies the region type.
Value: Specify one of the following:

FENCE	All instances assigned to this type of region must be exclusively placed inside the region boundaries. No other instances are allowed inside this region.
GUIDE	All instances assigned to this type of region should be placed inside this region, but it is a preference, not a hard constraint. Other constraints, such as wire length and timing can override it.

Regions Example

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

```
int regionCB (defwCallbackType_e type,
                   defiUserData userData) {
         int    res;

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

         res = defwStartRegions(1);
         CHECK_RES(res);
         res = defwRegionName("region2");
         CHECK_RES(res);
         res = defwRegionPoints(4000, 0, 5000, 1000);
         CHECK_RES(res);
         res = defwStringProperty("scum", "on bottom");
         CHECK_RES(res);
         res = defwEndRegions();
         CHECK_RES(res);

         return 0;}
```

Rows

The Row routines write a DEF `ROWS` statement. The `ROWS` statement is optional and can be used more than once in a DEF file. For syntax information about the DEF `ROWS` statement, see "Rows" in the *LEF/DEF Language Reference*.

If the DEF file contains a `DIEAREA` statement, the `ROWS` statement must follow it. For more information about the DEF `DIEAREA` writer routines, see "Die Area" on page 136.

Note: To write a `PROPERTY` statement for the row, you must use one of the property routines immediately following the `defwRow` routine. For more information, see "Property Statements" on page 200.

All routines return `0` if successful.

defwRow

Writes a `ROWS` statement.

Syntax

```
int defwRow(
     const char* rowName,
     const char* rowType,
     int origX,
     int origY,
     int orient,
     int do_count,
     int do_increment,
     int xstep,
     int ystep)
```

Arguments

do_count

Optional argument that specifies the number of columns in the array pattern. Specify `0` to ignore this argument.

do_increment

Optional argument that specifies the number of rows in the array pattern. Specify `0` to ignore this argument.

orient

Specifies the orientation of all sites in the row.
Value: 0 to 7. For more information, see "Orientation Codes" on page 19

rowName

Specifies the row name for this row.

rowType

Specifies the site to use for the row.

stepX stepY

Optional arguments that specify the spacing between the columns and rows. Specify `0` to ignore these arguments.

x_orig y_orig

Specifies the location in the design of the first site in the row.

defwRowStr

Also writes a `ROWS` statement. This routine is the same as the `defwRow` routine, with the exception of the *orient* argument, which takes a string instead of an integer.

Syntax

```
int defwRowStr (
     const char* rowName,
     const char* rowType,
     int x_orig,
     int y_orig,
     const char* orient,
     int do_count,
     int do_increment,
     int xstep,
     int ystep)
```

Arguments

do_count

Optional argument that specifies the number of columns in the array pattern. Specify `0` to ignore this argument.

do_increment

Optional argument that specifies the number of rows in the array pattern. Specify `0` to ignore this argument.

orient

Specifies the orientation of all sites in the row.
Value: `N`, `W`, `S`, `E`, `FN`, `FW`, `FS`, or `FE`

rowName

Specifies the row name for this row.

rowType

Specifies the site to use for the row.

stepX stepY

Optional argument that specifies the spacing between the columns and rows. Specify `0` to ignore these arguments.

x_orig y_orig

Specifies the location in the design of the first site in the row.

Rows Example

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

```
int rowCB (defwCallbackType_e type,
                defiUserData userData) {
         int    res;

         nt regionCB (defwCallbackType_e type,
                   defiUserData userData) {
         int    res;
         // Check if the type is correct
         if (type != defwRowCbkType) {
             printf("Type is not defwRowCbkType, terminate
                writing.\n");
             return 1;
         }

         res = defwRow("ROW_9", "CORE", -177320, -111250, 5, 911, 1,
                              360, 0);
         CHECK_RES(res);
         res = defwRealProperty("minlength", 50.5);
         CHECK_RES(res);
         res = defwStringProperty("firstName", "Only");
```

```
CHECK_RES(res);
res = defwIntProperty("idx", 1);
CHECK_RES(res);
res = defwRow("ROW_10", "CORE1", -19000, -11000, 6, 1, 100,
              0, 600);
CHECK_RES(res);

return 0;}
```

Scan Chains

The Scan Chain routines write a DEF `SCANCHAINS` statement. The `SCANCHAINS` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `SCANCHAINS` statement, see "Scan Chains" in the *LEF/DEF Language Reference*.

You must begin and end a DEF `SCANCHAINS` statement with the `defwStartScanchains` and `defwEndScanchains` routines. You must define all scan chains between these routines. Each scan chain specification must start with a `defwScanchains` routine.

For examples of the routines described here, see "Scan Chain Example" on page 216.

Note: To write a `PROPERTY` statement for the region, you must use one of the property routines following `defwScanchains`. For more information, see "Property Statements" on page 200.

All routines return `0` if successful.

defwStartScanchains

Starts the `SCANCHAINS` statement.

Syntax

```
int defwStartScanchains(
     int count)
```

Arguments

count

Specifies the number of scan chains defined in the `SCANCHAINS` statement.

defwEndScanchains

Ends the `SCANCHAINS` statement. If *count* specified in the `defwStartScanChains` routine is not the same as the actual number of `defwScanChain` routines used, this routine returns `DEFW_BAD_DATA`.

Syntax

```
int defwEndScanchains()
```

defwScanchain

Starts a scan chain specification. This routine must be used the number of times specified in the `defwStartScanchains` *count* argument.

Syntax

```
int defwScanchain(
     const char* chainName)
```

Arguments

chainName

Specifies the name of the scan chain.

defwScanchainCommonscanpins

Writes a `COMMONSCANPINS` statement. The `COMMONSCANPINS` statement is optional and can be used only once for each scan chain.

Syntax

```
int defwScanchainCommonscanpins(
     const char* inst1,
     const char* pin1,
     const char* inst2,
     const char* pin2)
```

Arguments

inst1 inst2

Optional arguments that specify the common scan-in and scan-out pins. The *inst1* argument can have the value `IN` or `OUT`. The *inst2* argument can have the remaining `IN` or `OUT` value not specified in the *inst1* argument. Specify `NULL` to ignore either of these arguments.

pin1 pin2

Specifies the names of the scan-in and scan-out pins that correspond with the value of *inst1* and *inst2*. Specify `NULL` to ignore either of these arguments.

Note: The *inst1/pin1* and *inst2/pin2* arguments must be used as pairs. If you specify `NULL` for either *inst1* or *inst2*, you must also specify `NULL` for the corresponding *pin1* or *pin2*. Similarly, if you specify `IN` or `OUT` for *inst1* or *inst2*, you must specify a pin name for the corresponding *pin1* or *pin2*.

defwScanchainFloating

Writes a `FLOATING` statement. The `FLOATING` statement is optional and can be used more than once for each scan chain.

Syntax

```
int defwScanchainFloating(
     const char* floatingComp,
     const char* inst1,
     const char* pin1,
     const char* inst2,
     const char* pin2)
```

Arguments

floatingComp

Specifies the floating component name.

inst1 inst2

Optional arguments that specify the in and out pins for the component. The *inst1* argument can have the value `IN` or `OUT`. The *inst2* argument can have the remaining `IN` or `OUT` value not specified in the *inst1* argument. Specify `NULL` to ignore either of these arguments.

pin1 pin2

Specifies the names of the in and out pins that correspond with the value of *inst1* and *inst2*. Specify NULL to ignore either of these arguments.

Note: The *inst1/pin1* and *inst2/pin2* arguments must be used as pairs. If you specify NULL for either *inst1* or *inst2*, you must also specify NULL for the corresponding *pin1* or *pin2*. Similarly, if you specify IN or OUT for *inst1* or *inst2*, you must specify a pin name for the corresponding *pin1* or *pin2*.

defwScanchainFloatingBits

Writes a FLOATING statement that contains BITS information. The FLOATING statement is optional and can be used more than once for each scan chain.

Syntax

```
int defwScanchainFloatingBits(
     const char* floatingComp,
     const char* inst1,
     const char* pin1,
     const char* inst2,
     const char* pin2,
     int bits)
```

Arguments

floatingComp

Specifies the floating component name.

inst1 inst2

Optional arguments that specify the in and out pins for the component. The *inst1* argument can have the value IN or OUT. The *inst2* argument can have the remaining IN or OUT value not specified in the *inst1* argument. Specify NULL to ignore either of these arguments.

pin1 pin2

Specifies the names of the in and out pins that correspond with the value of *inst1* and *inst2*. Specify NULL to ignore either of these arguments.

Note: The *inst1/pin1* and *inst2/pin2* arguments must be used as pairs. If you specify NULL for either *inst1* or *inst2*, you must also specify NULL for the corresponding *pin1* or *pin2*. Similarly, if you specify `IN` or `OUT` for *inst1* or *inst2*, you must specify a pin name for the corresponding *pin1* or *pin2*.

bits

Optional argument that specifies the sequential bit length of any chain element. Specify `-1` to ignore this argument.

defwScanchainOrdered

Writes an `ORDERED` statement. The `ORDERED` statement specifies an ordered list of scan chains. The `ORDERED` statement is optional and can be used more than once for each scan chain.

Syntax

```
int defwScanchainOrdered(
     const char* name1,
     const char* inst1,
     const char* pin1,
     const char* inst2,
     const char* pin2,
     const char* name2,
     const char* inst3,
     const char* pin3,
     const char* inst4,
     const char* pin4)
```

Arguments

name1 name2

Specifies the fixed component names. You must specify both *name1* and *name2* the first time you call this routine within a scanchain. If you call this routine multiple times within a scanchain, you only need to specify *name1*.

inst1 inst2 inst3 inst4

Optional arguments that specify the scan-in and scan-out pins for the components. The *inst1* and *inst3* arguments can have the value `IN` or `OUT`. The *inst2* and *inst4* arguments can have the remaining `IN` or `OUT` not specified in the *inst1* or *inst3* arguments. Specify `NULL` to ignore any of these arguments.

pin1 pin2 pin3 pin4

Specifies the names of the scan-in and scan-out pins that correspond with the *inst** values. Specify `NULL` to ignore any of these arguments.

Note: The *inst*/pin** arguments must be used as pairs. If you specify `NULL` for *inst1*, you must also specify `NULL` for the corresponding *pin1*. Similarly, if you specify `IN` or `OUT` for *inst1*, you must specify a pin name for the corresponding *pin1*.

defwScanchainOrderedBits

Writes an `ORDERED` statement that contains `BITS` information. The `ORDERED` statement specifies an ordered list of scan chains. The `ORDERED` statement is optional and can be used more than once for each scan chain.

Syntax

```
int defwScanchainOrderedBits(
     const char* name1,
     const char* inst1,
     const char* pin1,
     const char* inst2,
     const char* pin2,
     int bits1,
     const char* name2,
     const char* inst3,
     const char* pin3,
     const char* inst4,
     const char* pin4,
     int bits2)
```

Arguments

name1 name2

Specifies the fixed component names. You must specify both *name1* and *name2* the first time you call this routine within a scanchain. If you call this routine multiple times within a scanchain, you only need to specify *name1*.

inst1 inst2 inst3 inst4

Optional arguments that specify the scan-in and scan-out pins for the components. The *inst1* and *inst3* arguments can have the value `IN` or `OUT`. The *inst2* and *inst4* arguments can have the remaining `IN` or `OUT` not specified in the *inst1* or *inst3* arguments. Specify `NULL` to ignore any of these arguments.

pin1 pin2 pin3 pin4

Specifies the names of the scan-in and scan-out pins that correspond with the *inst** values. Specify `NULL` to ignore any of these arguments.
Note: The *inst*/pin** arguments must be used as pairs. If you specify `NULL` for

inst1, you must also specify `NULL` for the corresponding *pin1*. Similarly, if you specify `IN` or `OUT` for *inst1*, you must specify a pin name for the corresponding *pin1*.

*bits**

Optional argument that specifies the sequential bit length of any chain element. Specify `-1` to ignore this argument.

defwScanchainPartition

Writes a `PARTITION` statement. The `PARTITION` statement is optional and can be used only once to define a scan chain.

Syntax

```
int defwScanchainPartition(
     const char* name,
     int maxBits)
```

Arguments

name

Specifies a partition name. A partition name associates each chain with a partition group, which determines their compatibility for repartitioning by swapping elements between them. Chains with matching `PARTITION` names constitute a swap-compatible group.

maxBits

Optional argument that specifies the maximum bit length that the chain can grow to in the partition. Specify `-1` to ignore this argument.

defwScanchainStart

Writes a `START` statement. The `START` statement is required and can be used only once to define a scan chain.

Syntax

```
int defwScanchainStart(
     const char* inst,
     const char* pin)
```

Arguments

inst

Specifies the start of the scan chain. You can specify a component name, or the keyword `PIN` to specify an I/O pin.

pin

Specifies the out pin name. If you do not specify the out pin, DEF uses the out pin specified for common scan pins. If the scan chain starts at an I/O pin, you must specify the I/O pin name as the out pin.

defwScanchainStop

Writes a `STOP` statement. The `STOP` statement is required and can be used only once to define a scan chain.

Syntax

```
int defwScanchainStop(
     const char* inst,
     const char* pin)
```

Arguments

inst

Specifies the end point of the scan chain. You can specify a component name, or the keyword `PIN` to specify an I/O pin.

pin

Specifies the in pin name. If you do not specify the in pin, DEF uses the in pin specified for common scan pins. If the scan chain starts at an I/O pin, you must specify the I/O pin name as the in pin.

Scan Chain Example

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

```
int scanchainCB (defwCallbackType_e type,
                      defiUserData userData) {
          int    res;

          // Check if the type is correct
```

```
    if (type != defwScanchainCbkType) {
        printf("Type is not defwScanchainCbkType, terminate
          writing.\n");
        return 1;
    }


     res = defwStartScanchains(1);
    CHECK_RES(res);
    res = defwScanchain("the_chain");
    CHECK_RES(res);
    res = defwScanchainCommonscanpins("IN", "PA1", "OUT", "PA2")
    CHECK_RES(res);
    res = defwScanchainStart("PIN", "scanpin");
    CHECK_RES(res);
    res = defwScanchainStop("cell4", "PA2");
    CHECK_RES(res);
    res = defwScanchainOrdered("cell2", "IN", "PA0", NULL
                               NULL, "cell1", "OUT", "P10", NULL,
                               NULL);
    CHECK_RES(res);
    res = defwScanchainFloating("scancell1", "IN", "PA0",
                               NULL, NULL)
    CHECK_RES(res);
    res = defwEndScanchain();
    CHECK_RES(res);

  return 0;}
```

Special Nets

Special Nets routines write a DEF `SPECIALNETS` statement. The `SPECIALNETS` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `SPECIALNETS` statement, see "Special Nets" in the *LEF/DEF Language Reference*.

A `SPECIALNETS` statement must start and end with the `defwStartSpecialNets` and `defwEndSpecialNets` routines. All special nets must be defined between these routines. Each individual special net must start and end with the `defwSpecialNet` and `defwSpecialNetEndOneNet` routines.

For examples of the routines described here, see "Special Nets Example" on page 224.

In addition to the routines in this section, you can also include routines that form a *specialWiring* statement and a `PROPERTY` statement. For information about these routines, see "Special Wiring" on page 225 and "Property Statements" on page 200.

All routines return `0` if successful.

defwStartSpecialNets

Starts the `SPECIALNETS` statement.

Syntax

```
int defwStartSpecialNets(
     int count)
```

Arguments

count

Specifies the number of special nets defined in the `SPECIALNETS` statement.

defwEndSpecialNets

Ends the `SPECIALNETS` statement. If *count* specified in `defwStartSpecialNets` is not the same as the actual number of `defwSpecialNet` routines used, this routine returns `DEFW_BAD_DATA`.

Syntax

```
int defwEndSpecialNets()
```

defwSpecialNet

Starts a special net description. Each special net in the `SPECIALNETS` statement must start and end with `defwSpecialNet` and `defwSpecialNetEndOneNet`.

Syntax

```
int defwSpecialNet(
     const char* netName)
```

Arguments

netName

Specifies the name of the net to define.

defwSpecialNetEndOneNet

Ends the special net description started with `defwSpecialNet`. Each special net in the `SPECIALNETS` statement must start and end with `defwSpecialNet` and `defwSpecialNetEndOneNet`.

Syntax

```
int defwSpecialNetEndOneNet()
```

defwSpecialNetConnection

Specifies the special pin and component information for the special net. This routine is optional and can be used only once for each special net in the `SPECIALNETS` statement.

Syntax

```
int defwSpecialNetConnection(
     const char* compNameRegExpr,
     const char* pinName,
     int synthesized)
```

Arguments

compNameRegExpr

Specifies a component name or a regular expression that specifies a set of component names.

pinName

Specifies the name of the special pin on the net that corresponds to the component. During evaluation of the regular expression, components that match the expression but do not have a pin named *pinName* are ignored.

synthesized

Optional argument that marks the pin as part of a synthesized scan chain.
Value: Specify one of the following:

0	Argument is ignored.
1	Writes a `SYNTHESIZED` statement.

defwSpecialNetEstCap

Writes an `ESTCAP` statement. The `ESTCAP` statement is optional and can be used only once for each special net in the `SPECIALNETS` statement.

Syntax

```
int defwSpecialNetEstCap(
     double wireCap)
```

Arguments

wireCap

Specifies the estimated wire capacitance for the net. `ESTCAP` can be loaded with simulation data to generate net constraints for timing-driven layout.

defwSpecialNetFixedBump

Writes a `FIXEDBUMP` statement that indicates the bump cannot be assigned to a different pin. The `FIXEDBUMP` statement is optional and can be used only once for each special net in the `SPECIALNETS` statement.

Syntax

```
defwSpecialNetFixedBump()
```

defwSpecialNetOriginal

Writes an `ORIGINAL` statement. The `ORIGINAL` statement is optional and can be used only once for each special net in the `SPECIALNETS` statement.

Syntax

```
int defwSpecialNetOriginal(
     const char* netName)
```

Arguments

netName

Specifies the original net partitioned to create multiple nets, including the current net.

defwSpecialNetPattern

Writes a `PATTERN` statement. The `PATTERN` statement is optional and can be used only once for each special net in the `SPECIALNETS` statement.

Syntax

```
int defwSpecialNetPattern(
     const char* name)
```

Arguments

name

Specifies the routing pattern used for the net.
Value: Specify one of the following:

`BALANCED`	Used to minimize skews in timing delays for clock nets.
`STEINER`	Used to minimize net length.
`TRUNK`	Used to minimize delay for global nets.
`WIREDLOGIC`	Used in ECL designs to connect output and mustjoin pins before routing to the remaining pins.

defwSpecialNetSource

Writes a `SOURCE` statement. The `SOURCE` statement is optional and can only be used once for each special net in the `SPECIALNETS` statement.

Syntax

```
int defwSpecialNetSource(
     const char* name)
```

Arguments

name

Specifies the source of the net.
Value: Specify one of the following:

DIST	Net is the result of adding physical components (that is, components that only connect to power or ground nets), such as filler cells, well-taps, tie-high and tie-low cells, and decoupling caps.
NETLIST	Net is defined in the original netlist. This is the default value, and is not normally written out in the DEF file.
TEST	Net is part of a scanchain.
TIMING	Net represents a logical rather than physical change to netlist, and is used typically as a buffer for a clock-tree, or to improve timing on long nets.
USER	Net is user defined.

defwSpecialNetUse

Writes a USE statement. The USE statement is optional and can be used only once for each special net in the SPECIALNETS statement.

Syntax

```
int defwSpecialNetUse(
     const char* name)
```

Arguments

name

Specifies how the net is used.
Value: Specify one of the following:

ANALOG	Used as a analog signal net.
CLOCK	Used as a clock net.

`GROUND`	Used as a ground net.
`POWER`	Used as a power net.
`RESET`	Used as a reset net.
`SCAN`	Used as a scan net.
`SIGNAL`	Used as digital signal net.
`TIEOFF`	Used as a tie-high or tie-low net.

defwSpecialNetVoltage

Writes a `VOLTAGE` statement. The `VOLTAGE` statement is optional and can be used only once for each special net in the `SPECIALNETS` statement.

Syntax

```
int defwSpecialNetVoltage(
     double volts)
```

Arguments

`volts`

Specifies the voltage for the net as an integer in units of .001 volts. For Example, 1.5 v is equal to 1500 in DEF.

defwSpecialNetWeight

Writes a `WEIGHT` statement. The `WEIGHT` statement is optional and can be used only once for each special net in the `SPECIALNETS` statement.

Syntax

```
int defwSpecialNetWeight(
     double weight)
```

Arguments

weight

Specifies the weight of the net. Automatic layout tools attempt to shorten the lengths of nets with high weights. Do not specify a net weight larger than `10`, or assign weights to more than `3` percent of the nets in a design.

Special Nets Example

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

```
int snetCB (defwCallbackType_e type,
                 defiUserData userData) {
         int    res;
         const char **coorX, **coorY;


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

          res = defwStartSpecialNets(2);
         CHECK_RES(res);
         res = defwSpecialNet("net1");
         CHECK_RES(res);
         res = defwSpecialNetConnection("cell1", "VDD", 0);
         CHECK_RES(res);
         res = defwSpecialNetWidth("M1", 200);
         CHECK_RES(res);
         res = defwSpecialNetVoltage(3.2);
         CHECK_RES(res);
         res = defwSpecialNetSpacing("M1", 200, 190, 210);
         CHECK_RES(res);
         res = defwSpecialNetSource("TIMING");
         CHECK_RES(res);
         res = defwSpecialNetOriginal("VDD");
         CHECK_RES(res);
         res = defwSpecialNetUse("POWER");
         CHECK_RES(res);
         res = defwSpecialNetWeight(30);
         CHECK_RES(res);
         res = defwStringProperty("contype", "star");
         CHECK_RES(res);
         res = defwIntProperty("ind", 1);
         CHECK_RES(res);
```

```
res = defwRealProperty("maxlength", 12.13);
CHECK_RES(res);
res = defwSpecialNetEndOneNet();
CHECK_RES(res);
res = defwSpecialNet("VSS");
CHECK_RES(res);
res = defwSpecialNetConnection("cell1", "GND", 0);
CHECK_RES(res);

 ...
// An example on Special Wiring can be found under the
// Special Wiring section.

...
// An example on SpecialNet Shield can be found under the
// Shielded Routing section.

 res = defwSpecialNetPattern("STEINER");
CHECK_RES(res);
res = defwSpecialNetEstCap(100);
CHECK_RES(res);
res = defwSpecialNetEndOneNet();
CHECK_RES(res);
res = defwEndSpecialNets();
CHECK_RES(res);
return 0;}
```

Special Wiring

Special wiring routines form a *specialWiring* statement that can be used to define the wiring for both routed and shielded nets.The *specialWiring* statement is optional and can be used more than once in a `SPECIALNET` statement. For syntax information about the DEF `SPECIALNETS` statement, see "Special Nets" in the *LEF/DEF Language Reference*.

A *specialWiring* statement can include routines to define either rectangles, polygons, or a path of points to create the routing for the nets. Each path of points must start and end with the `defwSpecialNetPathStart` and `defwSpecialNetPathEnd` routines. If defined, a *`specialWiring`* statement must be included between the `defwSpecialNet` and `defwEndOneNet` routines.

For examples of the routines described here, see "Special Wiring Example" on page 232.

All routines return `0` if successful.

defwSpecialNetPathStart

Starts a *specialWiring* statement. Each *`specialWiring`* statement must start and end with `defwSpecialNetPathStart` and `defwSpecialNetPathEnd`.

Syntax

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

Arguments

`type`

Specifies the special wiring type. If no wiring is specified for a particular net, the net is unrouted.
Value: Specify one of the following:

`COVER`	Specifies that the wiring cannot be moved by either automatic layout or interactive commands.
`FIXED`	Specifies that the wiring cannot be moved by automatic layout, but can be changed by interactive commands.
`ROUTED`	Specifies that the wiring can be moved by automatic layout tools.
`SHIELD`	Specifies that the special net being defined shields a regular net.
`NEW`	Indicates a new wire segment.

defwSpecialNetPathEnd

Ends the *specialWiring* statement. Each *`specialWiring`* statement must start and end with `defwSpecialNetPathStart` and `defwSpecialNetPathEnd`.

Syntax

```
int defwSpecialNetPathEnd()
```

defwSpecialNetPathLayer

Writes a `LAYER` statement. Either a `LAYER`, `POLYGON`, or `RECT` statement is required for each specialWiring statement.The `LAYER` statement can be used more than once for each *`specialWiring`* statement.

Syntax

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

Arguments

`layerName`

Specifies the layer on which the wire lies.

defwSpecialNetPathPoint

Defines the center line coordinates of the route on the layer specified with `defwSpecialNetPathLayer`. Either this routine or `defwSpecialNetPathPointWithWireExt` is required with a `LAYER` statement, and can be used only once for each `LAYER` statement in a *`specialWiring`* statement.

Syntax

```
int defwSpecialNetPathPoint(
     int numPts,
     const char** pointX,
     const char** pointY)
```

Arguments

`numPts`

Specifies the number of points in the route.

`pointX pointY`

Specifies the route coordinates.

defwSpecialNetPathPointWithWireExt

Defines the center line coordinates and wire extension value of the route on the layer specified with `defwSpecialNetPathLayer`. Either this routine or `defwSpecialNetPathPoint` is required with a `LAYER` statement, and can be used only once for each `LAYER` statement in a specialWiring statement.

Syntax

```
defwSpecialNetPathPointWithWireExt(
     int numPoints,
     const char** pointX,
     const char** pointY,
     const char** value)
```

Arguments

numPoints

Specifies the number of points in the route.

pointX pointY

Specifies the route coordinates.

value

Optional argument that specifies the amount by which the wire is extended past the endpoint of the segment. Specify `NULL` to ignore this argument.

defwSpecialNetPathShape

Writes a `SHAPE` statement. The `SHAPE` statement is optional with a `LAYER` statement, and can be used only once for each `LAYER` statement in a *specialWiring* statement.

Syntax

```
int defwSpecialNetPathShape(
     const char* shapeType)
```

Arguments

shapeType

Specifies a wire with special connection requirements because of its shape.
Value: RING, PADRING, BLOCKRING, STRIPE, FOLLOWPIN, IOWIRE, COREWIRE, BLOCKWIRE, FILLWIRE, BLOCKAGEWIRE, or DRCFILL

defwSpecialNetPathStyle

Writes a STYLE statement. A STYLE statement is optional with a LAYER statement, and can be used only once for each LAYER statement in a *specialWiring* statement.

Syntax

```
defwSpecialNetStyle(
     int styleNum)
```

Arguments

styleNum

Specifies a previously defined style number from the STYLES section in this DEF file.

defwSpecialNetPathVia

Specifies a via for the special wiring. This routine is optional with a LAYER statement, and can be used only once for each LAYER statement in a *specialWiring* statement.

Syntax

```
int defwSpecialNetPathVia(
     const char* viaName)
```

Arguments

viaName

Specifies a via to place at the last point of the route.

defwSpecialNetPathViaData

Creates an array of power vias of the via specified with `defwSpecialNetPathVia`. This routine is optional with a `LAYER` statement, and can be used only once for each `LAYER` statement in a *`specialWiring`* statement.

Syntax

```
int defwSpecialNetPathViaData(
     int numX,
     int numY,
     int stepX,
     int stepY)
```

Arguments

`numX numY`

Specifies the number of vias to create in the x and y directions.

`stepX stepY`

Specifies the step distance between vias, in the x and y directions

defwSpecialNetPathWidth

Writes a `WIDTH` statement. The `WIDTH` statement is required with a `LAYER` statement, and can be used only once for each `LAYER` statement in a *`specialWiring`* statement.

Syntax

```
int defwSpecialNetPathWidth(
     int width)
```

Arguments

`width`

Specifies the width for wires on the layer specified with `defwSpecialNetPathLayer`.

defwSpecialNetShieldNetName

Specifies the name of a regular net to be shielded by the special net being defined. This routine is required if `SHIELD` is specified in the `defwSpecialNetPathStart` routine and can be used only once for each *`specialWiring`* statement.

Syntax

```
int defwSpecialNetShieldNetName(
     const char* name)
```

Arguments

`name`

Specifies the name of the regular net to be shielded.

defwSpecialNetPolygon

Writes a `POLYGON` statement. Either a `LAYER`, `POLYGON`, or `RECT` statement is required for each *`specialWiring`* statement. The `POLYGON` statement can be used only once for each *`specialWiring`* statement.

Syntax

```
defwSpecialNetPolygon(
     const char* layerName,
     int num_polys,
     double* xl,
     double* yl)
```

Arguments

`layerName`

Specifies the layer on which to generate the polygon.

`num_polys`

Specifies the number of polygon sides.

`xl yl`

Specifies a sequence of points to generate a polygon geometry on *`layerName`*. The polygon edges must be parallel to the x axis, the y axis, or at a 45-degree angle.

defwSpecialNetRect

Writes a RECT statement. Either a LAYER, POLYGON, or RECT statement is required for each specialWiring statement. The RECT statement can be used only once for each *specialWiring* statement.

Syntax

```
defwSpecialNetRect(
     const char* layerName,
     int xl,
     int yl,
     int xh,
     int yh)
```

Arguments

layerName

Specifies the layer on which to create the rectangle.

xl yl xh yh

Specifies the coordinates of two points which define the opposite corners of the rectangle.

Special Wiring Example

The following example only shows the usage of some functions related to special wiring in a special net. This example is part of the special net callback routine.

```
int snetCB (defwCallbackType_e type,
                  defiUserData userData) {
          int    res;
          const char **coorX, **coorY;


           ...
          res = defwSpecialNetPathStart("ROUTED");
          CHECK_RES(res);
          res = defwSpecialNetPathLayer("M1");
          CHECK_RES(res);
          res = defwSpecialNetPathWidth(250);
          CHECK_RES(res);
          res = defwSpecialNetPathShape("IOWIRE");
          CHECK_RES(res);
          coorX = (const char**)malloc(sizeof(char*)*3);
```

```
coorY = (const char**)malloc(sizeof(char*)*3);
coorX[0] = strdup("5");
coorY[0] = strdup("15");
coorX[1] = strdup("125");
coorY[1] = strdup("*");
coorX[2] = strdup("245");
coorY[2] = strdup("*");
res = defwSpecialNetPathPoint(3, coorX, coorY);
CHECK_RES(res);
res = defwSpecialNetPathEnd();
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
...

return 0;}
```

Shielded Routing

The shielded routing routines form a *shielded routing* specification that can be used to define a special net. The *shielded routing* specification is optional and can be used more than once in a `SPECIALNET` statement. For syntax information about the DEF `SPECIALNETS` statement, see Special Nets in the *LEF/DEF Language Reference*.

You must begin and end a *shielded routing* specification with the `defwSpecialNetShieldStart` and `defwSpecialNetShieldEnd` routines. You must define all shielded routing between these routines. The shielded routing routines must be included between the `defwSpecialNet` and `defwEndOneNet` routines.

For examples of the routines described here, see "Shielded Routing Example" on page 236.

defwSpecialNetShieldStart

Starts the shielded routing specification. This routine is optional and can be used only once to define each special net shield.

Syntax

```
int defwSpecialNetShieldStart(
     const char* name)
```

Arguments

name

Specifies the net shield name.

defwSpecialNetShieldEnd

Ends the shielded routing specification.

Syntax

```
int defwSpecialNetShieldEnd()
```

defwSpecialNetShieldLayer

Writes a `LAYER` statement. The `LAYER` statement is required and can be used only once per special net shield.

Syntax

```
int defwSpecialNetShieldLayer(
     const char* name)
```

Arguments

name

Specifies the layer on which the wire lies.

defwSpecialNetShieldPoint

Specifies the points of the wire path in the special net shield. This routine is optional and can be used more than once per special net shield.

Syntax

```
int defwSpecialNetShieldPoint(
     int numPts,
     const char** pointx,
     const char** pointy)
```

Arguments

numPts

Specifies the number of points in the special net shield.

pointx pointy

Specifies the coordinate locations for the path points.

defwSpecialNetShieldShape

Writes a `SHAPE` statement. The `SHAPE` statement is optional and can be used only once per special net shield.

Syntax

```
int defwSpecialNetShieldShape(
     const char* shapeType)
```

Arguments

shapeType

Specifies a wire with special connection requirements because of its shape.
Value: `RING`, `PADRING`, `BLOCKRING`, `STRIPE`, `FOLLOWPIN`, `IOWIRE`, `COREWIRE`, `BLOCKWIRE`, `FILLWIRE`, or `BLOCKAGEWIRE`

defwSpecialNetShieldVia

Specifies a via name for the special net shield. This routine is optional and can be used more than once per special net shield.

Syntax

```
int defwSpecialNetShieldVia(
     const char* name)
```

Arguments

name

Specifies the via to place at the last specified path coordinate.

defwSpecialNetShieldViaData

Creates an array of power vias of the via specified with the `defwSpecialNetShieldVia` routine. This routine is optional and can be used more than once for a special net.

Syntax

```
int defwSpecialNetShieldViaData(
      int numX,
      int numY,
      int stepX,
      int stepY)
```

Arguments

numX numY

Specifies the number of vias to create in the x and y directions.

stepX stepY

Specifies the step distance in the x and y directions.

defwSpecialNetShieldWidth

Writes a `WIDTH` statement. The `WIDTH` statement is required and can be used only once per special net shield.

Syntax

```
int defwSpecialNetShieldWidth(
      int width)
```

Arguments

width

Specifies the wire width.

Shielded Routing Example

The following example only shows the usage of some functions related to shielded routing in a special net. This example is part of the special net callback routine.

```
int snetCB (defwCallbackType_e type,
                defiUserData userData) {
         int    res;
         const char **coorX, **coorY;

         ...
         res = defwSpecialNetShieldStart("my_net");
         CHECK_RES(res);
         res = defwSpecialNetShieldLayer("M2");
         CHECK_RES(res);
         res = defwSpecialNetShieldWidth(90);
         CHECK_RES(res);
         coorX[0] = strdup("14100");
         coorY[0] = strdup("342440");
         coorX[1] = strdup("13920");
         coorY[1] = strdup("*");
         res = defwSpecialNetShieldPoint(2, coorX, coorY);
         CHECK_RES(res);
         res = defwSpecialNetShieldVia("M2_TURN");
         CHECK_RES(res);
         free((char*)coorX[0]);
         free((char*)coorY[0]);
         coorX[0] = strdup("*");
         coorY[0] = strdup("263200");
         res = defwSpecialNetShieldPoint(1, coorX, coorY);
         CHECK_RES(res);
         res = defwSpecialNetShieldVia("M1_M2");
         CHECK_RES(res);
         free((char*)coorX[0]);
         free((char*)coorY[0]);
         coorX[0] = strdup("2400");
         coorY[0] = strdup("*");
         res = defwSpecialNetShieldPoint(1, coorX, coorY);
         CHECK_RES(res);
         res = defwSpecialNetShieldEnd();
         ...

         return 0;}
```

Slots

Slots routines write a DEF `SLOTS` statement. The `SLOTS` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `SLOTS` statement, see "Slots" in the *LEF/DEF Language Reference*.

The `SLOTS` statement must start and end with the `defwStartSlots` and `defwEndSlots` routines. All slots must be defined between these routines.

All routines return `0` if successful.

defwStartSlots

Starts a `SLOTS` statement.

Syntax

```
int defwStartSlots(
     int count)
```

Arguments

count

Specifies the number of `defwSlotLayer` routines in the `SLOTS` statement.

defwEndSlots

Ends the `SLOTS` statement.

Syntax

```
int defwEndSlots()
```

defwSlotLayer

Writes a `LAYER` statement. The `LAYER` statement is required for each slot and can be used more than once in a `SLOTS` statement.

Syntax

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

Arguments

layerName

Specifies the layer on which to create the slot.

defwSlotPolygon

Writes a `POLYGON` statement. Either a `POLYGON` or `RECT` statement is required with a `LAYER` statement. The `POLYGON` statement can be used more than once for each slot in the `SLOTS` statement.

Syntax

```
defwSlotPolygon(
     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.

defwSlotRect

Writes a `RECT` statement. The `RECT` statement is required and can be used more than once for each slot in the `SLOTS` statement.

Syntax

```
int defwSlotRect(
     int xl,
     int yl,
     int xh,
     int yh)
```

Arguments

xl yl xh yh

Specifies the coordinates of the slot geometry.

Styles

Styles routines write a DEF `STYLES` statement. The `STYLES` statement is optional and can be used only once in a DEF file. For syntax information about the `STYLES` statement, see "Styles" in the *LEF/DEF Language Reference*.

The `STYLES` statement must start and end with the `defwStartStyles` and `defwEndStyles` routines.

All routines return `0` if successful.

defwStartStyles

Starts the `STYLES` statement.

Syntax

```
defwStartStyles(
     int count)
```

Arguments

count

Specifies the number of styles defined in the `STYLES` statement.

defwEndStyles

Ends the `STYLES` statement.

Syntax

```
defwEndStyles()
```

defwStyles

Defines a style. This routine is required and can be used more than once in the `STYLES` statement.

Syntax

```
defwStyles(
     int styleNums,
     int num_points,
     double* xp,
     double* yp)
```

Arguments

styleNums

Defines a style. *styleNums* is a positive integer that is greater than or equal to 0 (zero), and is used to reference the style later in the DEF file. When defining multiple styles, the first *styleNums* must be 0 (zero), and any following *styleNums* should be numbered consecutively so that a table lookup can be used to find them easily.

num_points

Specifies the number of points in the style.

xp yp

Specifies a sequence of points to generate a polygon geometry. The syntax corresponds to a coordinate pair, such as *x y*. Specify an asterisk (*) to repeat the same value as the previous *x* or *y* value from the last point. The polygon must be convex. The polygon edges must be parallel to the x axis, the y axis, or at a 45-degree angle, and must enclose the point (0 0).

Technology

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

This routine returns 0 if successful.

defwTechnology

Writes a TECHNOLOGY statement.

Syntax

```
int defwTechnology(
     const char* technology)
```

Arguments

technology

Specifies a technology name for the design in the database.

Tracks

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

If the DEF file contains a `ROWS` statement, the `TRACKS` statement must follow it. For more information about the DEF `ROWS` writer routine, see “Rows” on page 206.

For examples of the routines described here, see “Tracks Example” on page 243.

This routine returns `0` if successful.

defwTracks

Writes a `TRACKS` statement.

Syntax

```
int defwTracks(
     const char* master,
     int doStart,
     int doCount,
     int doStep,
     int numLayers,
     const char** layers)
```

Arguments

doCount

Specifies the number of tracks to create.

doStep

Specifies the step spacing between the tracks.

doStart

Specifies the coordinate of the first line.

layers

Specifies the routing layers used for the tracks.

master

Specifies the direction for the first track defined.
Value: Specify one of the following:

`X`	Indicates vertical lines.
`Y`	Indicates horizontal lines.

numLayers

Specifies the number of routing layers to use for tracks.

Tracks Example

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

```
int trackCB (defwCallbackType_e type,
                  defiUserData userData) {
         int    res;
         const char** layers;

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


          layers = (const char**)malloc(sizeof(char*)*1);
         layers[0] = strdup("M1");
         res = defwTracks("X", 3000, 40, 120, 1, layers);
         CHECK_RES(res);
         free((char*)layers[0]);
         layers[0] = strdup("M2");
         res = defwTracks("Y", 5000, 10, 20, 1,layers);
         CHECK_RES(res);
         free((char*)layers[0]);
         free((char*)layers);
         res = defwNewLine();
```

```
CHECK_RES(res);

return 0;}
```

Units

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

This routine returns `0` if successful.

defwUnits

Writes a `UNITS` statement.

Syntax

```
int defwUnits(
     int units)
```

Arguments

`units`
    Specifies the convert factor used to convert DEF distance units into LEF distance units.

Version

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

This routine returns `0` if successful.

defwVersion

Writes a `VERSION` statement.

Syntax

```
int defwVersion(
     int vers1,
     int vers2)
```

Arguments

version1

Specifies the major number.

version2

Specifies the minor number.

Vias

Vias routines write a DEF `VIAS` statement. The `VIAS` statement is optional and can be used only once in a DEF file. For syntax information about the DEF `VIAS` statement, see "Vias" in the *LEF/DEF Language Reference*.

The `VIAS` statement must start and end with the `defwStartVias` and `defwEndVias` routines. All vias must be defined between these routines. Each individual via must start and end with the `defwViaName` and `defwOneViaEnd` routines.

For examples of the routines described here, see "Vias Example" on page 251.

All routines return `0` if successful.

defwStartVias

Starts a `VIAS` statement.

Syntax

```
int defwStartVias(
     int count)
```

Arguments

count

Specifies the number of vias defined in the `VIAS` statement.

defwEndVias

Ends the `VIAS` statement.

If the *`count`* specified in `defwStartVias` is not the same as the actual number of `defwViaName` routines used, this routine returns `DEFW_BAD_DATA`.

Syntax

```
int defwEndVias(void)
```

defwViaName

Starts a via description in the `VIAS` statement. Each via in the `VIAS` statement must start and end with `defwViaName` and `defwOneViaEnd`. This routine must be used the exact number of times specified with *`count`* in `defwStartVias`.

Each via can include one of the following routines:

- defwViaPolygon
- defwViaRect on page 247
- defwViaViarule on page 248

Syntax

```
int defwViaName(
     const char* name)
```

Arguments

`name`

Specifies the name of the via. Via names are generated by appending a number after the rule name. Vias are numbered in the order in which they are created.

defwOneViaEnd

Ends a via description in the `VIAS` statement. Each via in the `VIAS` statement must start and end with `defwViaName` and `defwOneViaEnd`. This routine must be used the exact number of times specified with *`count`* in `defwStartVias`.

Syntax

```
int defwOneViaEnd()
```

defwViaPolygon

Writes a `POLYGON` statement for a via in the VIAS statement. Either a `POLYGON`, `RECT`, or `VIARULE` statement can be specified for a via. The `POLYGON` statement is optional and can be used more than once for each via in the `VIAS` statement.

Syntax

```
int defwViaPolygon(
     const char* layerName,
     int num_polys,
     double* xl,
     double* yl)
```

Arguments

layerName

Specifies the layer on which to generate a polygon.

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, to the y axis, or at a 45-degree angle.

defwViaRect

Writes a `RECT` statement for a via in the `VIAS` statement. Either a `POLYGON`, `RECT`, or `VIARULE` statement can be specified for a via. The `RECT` statement is optional and can be used more than once for each via in the `VIAS` statement.

Syntax

```
int defwViaRect(
     const char* layerName,
     int xl,
```

```
     int yl,
     int xh,
     int yh)
```

Arguments

layerName

Specifies the layer on which the via geometry lies. All geometries for the via, including the cut layers, are output by the DEF writer.

xl yl xh yh

Defines the via geometry for the specified layer. The points are specified with respect to the via origin. In most cases, the via origin is the center of the via bounding box.

defwViaViarule

Writes a `VIARULE` statement for a via in the `VIAS` statement. Either a `POLYGON`, `RECT`, or `VIARULE` statement can be specified for a via. The `VIARULE` statement is optional and can be used only once for each via in the `VIAS` statement.

If you specify this routine, you can optionally specify the following routines:

- defwViaViaruleRowCol on page 249
- defwViaViaruleOrigin on page 250
- defwViaViaruleOffset on page 250
- defwViaViarulePattern on page 251

Syntax

```
defwViaViarule(
     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. The `VIARULE` must be a `VIARULE GENERATE` via rule; it cannot refer to a `VIARULE` without a `GENERATE` keyword.

xCutSize yCutSize

Specifies the required width (*xCutSize*) and height (*yCutSize*) of the cut layer rectangles.

botMetalLayer cutLayer topMetalLayer

Specifies the required names of the bottom routing layer, cut layer, and top routing layer. These layer names 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 form one cut edge to 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.

defwViaViaruleRowCol

Writes a `ROWCOL` statement in the `VIARULE` for a via. The `ROWCOL` statement is optional and can be used only once for each via in the `VIAS` statement.

Syntax

```
defwViaViaruleRowCol(
     int numCutRows,
     int numCutCols)
```

Arguments

numCutRows numCutCols

Specifies the number of cut rows and columns that make up the cut array.

defwViaViaruleOrigin

Writes an `ORIGIN` statement in a `VIARULE` statement for a via. The `ORIGIN` statement is optional and can be used only once for each via in the `VIAS` statement.

Syntax

```
defwViaViaruleOrigin(
     int xOffset,
     int 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.

defwViaViaruleOffset

Writes an `OFFSET` statement in a `VIARULE` statement for a via. The `OFFSET` statement is optional and can be used only once for each via in the `VIAS` statement.

Syntax

```
defwViaViaruleOffset(
     int xBotOffset,
     int yBotOffset,
     int xTopOffset,
     int yTopOffset)
```

Arguments

xBotOffset yBotOffset xTopOffset yTopOffset

Specifies the x and y offset for the bottom and top metal layers. These values allow each metal layer to be offset independently.
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, 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.

defwViaViarulePattern

Writes a `PATTERN` statement in a `VIARULE` statement for a via. The `PATTERN` statement is optional and can be used only once for each via in the `VIAS` statement.

Syntax

```
defwViaViarulePattern(
     const char* cutPattern)
```

Arguments

`cutPattern`
    Specifies the cut pattern encoded as an ASCII string.

Vias Example

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

```
int viaCB (defwCallbackType_e type,
                defiUserData userData) {
         int    res;

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

          res = defwStartVias(1);
         CHECK_RES(res);
         res = defwViaName("VIA_ARRAY");
         CHECK_RES(res);
         res = defwViaRect("M1", -40, -40, 40, 40);
         CHECK_RES(res);
         res = defwViaRect("V1", -40, -40, 40, 40);
         CHECK_RES(res);
         res = defwViaRect("M2", -50, -50, 50, 50);
         CHECK_RES(res);
         res = defwOneViaEnd();
         CHECK_RES(res);
         res = defwEndVias();
```

```
    CHECK_RES(res);

return 0;}
```

7

DEF Compressed File Routines

The Cadence® Design Exchange Format (DEF) reader provides the following routines for opening and closing compressed DEF files. These routines are used instead of the `fopen` and `fclose` routines that are used for regular DEF files.

- defGZipOpen on page 253
- defGZipClose on page 253
- Example on page 254

defGZipOpen

Opens a compressed DEF file. If the file opens with no errors, this routine returns a pointer to the file.

Syntax

```
defGZFile defGZipOpen(
     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"`.

defGZipClose

Closes the compressed DEF file. If the file closes with no errors, this routine returns zero.

Syntax

```
int defGZipClose(
     defGZFile filePtr) ;
```

Arguments

filePtr

Specifies a pointer to the compressed file to close.

Example

The following example uses the `defGZipOpen` and `defGZipClose` routines to open and close a compressed file.

```
defrInit() ;

for (fileCt = 0; fileCt < numInFile; fileCt++) {
    defrReset();
    // Open the compressed DEF file for the reader to read
    if ((f = defGZipOpen(inFile[fileCt], "r")) == 0) {
       fprintf(stderr, "Couldn't open input file '%s'\n", inFile[fileCt]);
       return(2) ;
    }
    // Set case sensitive to 0 to start with, in History and PropertyDefinition
    // reset it to 1.
    res = defrRead((FILE*)f, inFile[fileCt], (void*)userData, 1);

    if (res)
       fprintf(stderr, "Reader returns bad status.\n", inFile[fileCt]);

    // Close the compressed DEF file.
    defGZipClose(f);
    (void)defrPrintUnusedCallbacks(fout);
}
fclose(fout);

return 0;}
```

8

DEF File Comparison Utility

The Cadence® Design Exchange Format (DEF) reader provides the following utility for comparing DEF 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 DEF files:

```
#The names of the two DEF files that were compared.
< in.def
> out.def
#Statements listed under Deleted were found in in.def but not in out.def.
Deleted:
< BLOCKAGE LAYER m3 RECT 455 454 344 890
< BLOCKAGE LAYER m3 SLOTS
< BLOCKAGE LAYER m4 FILLS
< BLOCKAGE LAYER m4 RECT 455 454 344 890
< BLOCKAGE LAYER m5 PUSHDOWN
< BLOCKAGE LAYER m5 RECT 455 454 344 890
< BLOCKAGE PLACEMENT
Deleted:
< BLOCKAGE PLACEMENT PUSHDOWN
Deleted:
< BLOCKAGE PLACEMENT RECT 4000 6000 8000 4000
< BLOCKAGE PLACEMENT RECT 4000 6000 8000 4000
#Changed always contains two statements: the statement as it appears in in.def
and the statement as it appears in out.def.
Changed:
< COMP |i1 UNPLACED
< DESIGN muk
---
> DESIGN cell
Changed:
< NET net1 USE SCAN
---
> NET net1 WEIGHT 30 SOURCE TIMING ORIGINAL VDD USE SCAN
Changed:
< NET net3 SOURCE USER PATTERN BALANCED ORIGINAL extra_crispy USE SIGNAL
---
> NET net3 SOURCE USER PATTERN BALANCED ORIGINAL extra_crispy
#Statements listed under Added were found in out.def but not in in.def.
Added:
> NET SCAN ( PIN scanpin )
Added:
> NET net1 ( PIN pin1 )
Added:
> NET net2 ( PIN pin2 )
```

A

DEF Reader and Writer Examples

This appendix contains examples of the Cadence® Design Exchange Format (DEF) reader and writer.

- DEF Reader Example
- DEF Writer Example on page 325

DEF Reader Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#ifndef WIN32
#   include <unistd.h>
#endif /* not WIN32 */
#include "defrReader.hpp"
#include "defiAlias.hpp"

char defaultName[64];
char defaultOut[64];

// Global variables
FILE* fout;
int userData;
int numObjs;
int isSumSet;       // to keep track if within SUM
int isProp = 0;     // for PROPERTYDEFINITIONS
int begOperand;     // to keep track for constraint, to print - as the 1st char
static double curVer = 0;
static int setSNetWireCbk = 0;
```

```
// TX_DIR:TRANSLATION ON

void myLogFunction(const char* errMsg){
   fprintf(fout, "ERROR: found error: %s\n", errMsg);
}

void myWarningLogFunction(const char* errMsg){
   fprintf(fout, "WARNING: found error: %s\n", errMsg);
}

void dataError() {
  fprintf(fout, "ERROR: returned user data is not correct!\n");
}

void checkType(defrCallbackType_e c) {
  if (c >= 0 && c <= defrDesignEndCbkType) {
    // OK
  } else {
    fprintf(fout, "ERROR: callback type is out of bounds!\n");
  }
}


int done(defrCallbackType_e c, void* dummy, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "END DESIGN\n");
  return 0;
}

int endfunc(defrCallbackType_e c, void* dummy, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  return 0;
}


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");
}


int compf(defrCallbackType_e c, defiComponent* co, defiUserData ud) {
  int i;

  checkType(c);
  if ((long)ud != userData) dataError();
    fprintf(fout, "- %s %s ", co->defiComponent::id(),
            co->defiComponent::name());
    if (co->defiComponent::hasNets()) {
        for (i = 0; i < co->defiComponent::numNets(); i++)
             fprintf(fout, "%s ", co->defiComponent::net(i));
    }
    if (co->defiComponent::isFixed())
        fprintf(fout, "+ FIXED %d %d %s ",
                co->defiComponent::placementX(),
                co->defiComponent::placementY(),
                //orientStr(co->defiComponent::placementOrient()));
                co->defiComponent::placementOrientStr());
    if (co->defiComponent::isCover())
        fprintf(fout, "+ COVER %d %d %s ",
                co->defiComponent::placementX(),
                co->defiComponent::placementY(),
                orientStr(co->defiComponent::placementOrient()));
    if (co->defiComponent::isPlaced())
        fprintf(fout,"+ PLACED %d %d %s ",
                co->defiComponent::placementX(),
                co->defiComponent::placementY(),
                orientStr(co->defiComponent::placementOrient()));
    if (co->defiComponent::isUnplaced()) {
        fprintf(fout,"+ UNPLACED ");
        if ((co->defiComponent::placementX() != -1) ||
```

```
         (co->defiComponent::placementY() != -1))
        fprintf(fout,"%d %d %s ",
                co->defiComponent::placementX(),
                co->defiComponent::placementY(),
                orientStr(co->defiComponent::placementOrient()));
    }
    if (co->defiComponent::hasSource())
        fprintf(fout, "+ SOURCE %s ", co->defiComponent::source());
    if (co->defiComponent::hasGenerate()) {
        fprintf(fout, "+ GENERATE %s ", co->defiComponent::generateName());
        if (co->defiComponent::macroName() &&
            *(co->defiComponent::macroName()))
           fprintf(fout, "%s ", co->defiComponent::macroName());
    }
    if (co->defiComponent::hasWeight())
        fprintf(fout, "+ WEIGHT %d ", co->defiComponent::weight());
    if (co->defiComponent::hasEEQ())
        fprintf(fout, "+ EEQMASTER %s ", co->defiComponent::EEQ());
    if (co->defiComponent::hasRegionName())
        fprintf(fout, "+ REGION %s ", co->defiComponent::regionName());
    if (co->defiComponent::hasRegionBounds()) {
        int *xl, *yl, *xh, *yh;
        int size;
        co->defiComponent::regionBounds(&size, &xl, &yl, &xh, &yh);
        for (i = 0; i < size; i++) {
            fprintf(fout, "+ REGION %d %d %d %d \n",
                    xl[i], yl[i], xh[i], yh[i]);
        }
    }
    if (co->defiComponent::hasHalo()) {
        int left, bottom, right, top;
        (void) co->defiComponent::haloEdges(&left, &bottom, &right, &top);
        fprintf(fout, "+ HALO ");
        if (co->defiComponent::hasHaloSoft())
           fprintf(fout, "SOFT ");
        fprintf(fout, "%d %d %d %d\n", left, bottom, right, top);
    }
    if (co->defiComponent::hasRouteHalo()) {
        fprintf(fout, "+ ROUTEHALO %d %s %s\n", co->defiComponent::haloDist(),
                co->defiComponent::minLayer(), co->defiComponent::maxLayer());
    }
```

```
    if (co->defiComponent::hasForeignName()) {
        fprintf(fout, "+ FOREIGN %s %d %d %s %d ",
                co->defiComponent::foreignName(), co->defiComponent::foreignX(),
                co->defiComponent::foreignY(), co->defiComponent::foreignOri(),
                co->defiComponent::foreignOrient());
    }
    if (co->defiComponent::numProps()) {
        for (i = 0; i < co->defiComponent::numProps(); i++) {
            fprintf(fout, "+ PROPERTY %s %s ", co->defiComponent::propName(i),
                    co->defiComponent::propValue(i));
            switch (co->defiComponent::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");
    --numObjs;
    if (numObjs <= 0)
        fprintf(fout, "END COMPONENTS\n");
  return 0;
}


int netpath(defrCallbackType_e c, defiNet* ppath, defiUserData ud) {
  fprintf(fout, "\n");

  fprintf (fout, "Callback of partial path for net\n");

  return 0;
}
```

```
int netNamef(defrCallbackType_e c, const char* netName, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
    fprintf(fout, "- %s ", netName);
  return 0;
}

int subnetNamef(defrCallbackType_e c, const char* subnetName, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
    if (curVer >= 5.6)
      fprintf(fout, "   + SUBNET CBK %s ", subnetName);
  return 0;
}

int nondefRulef(defrCallbackType_e c, const char* ruleName, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
    if (curVer >= 5.6)
      fprintf(fout, "   + NONDEFAULTRULE CBK %s ", ruleName);
  return 0;
}

int netf(defrCallbackType_e c, defiNet* net, defiUserData ud) {
  // For net and special net.
  int        i, j, k, x, y, z, count, newLayer;
  defiPath*  p;
  defiSubnet *s;
  int        path;
  defiVpin   *vpin;
  // defiShield *noShield;
  defiWire   *wire;

  checkType(c);
  if ((long)ud != userData) dataError();
  if (c != defrNetCbkType)
      fprintf(fout, "BOGUS NET TYPE  ");
  if (net->defiNet::pinIsMustJoin(0))
      fprintf(fout, "- MUSTJOIN ");

  // compName & pinName
```

```
for (i = 0; i < net->defiNet::numConnections(); i++) {
    // set the limit of only 5 items per line
    count++;
    if (count >= 5) {
        fprintf(fout, "\n");
        count = 0;
    }
    fprintf(fout, "( %s %s ) ", net->defiNet::instance(i),
            net->defiNet::pin(i));
    if (net->defiNet::pinIsSynthesized(i))
        fprintf(fout, "+ SYNTHESIZED ");
}

if (net->hasNonDefaultRule())
    fprintf(fout, "+ NONDEFAULTRULE %s\n", net->nonDefaultRule());

for (i = 0; i < net->defiNet::numVpins(); i++) {
    vpin = net->defiNet::vpin(i);
    fprintf(fout, "  + %s", vpin->name());
    if (vpin->layer())
        fprintf(fout, " %s", vpin->layer());
    fprintf(fout, " %d %d %d %d", vpin->xl(), vpin->yl(), vpin->xh(),
            vpin->yh());
    if (vpin->status() != ' ') {
        fprintf(fout, " %c", vpin->status());
        fprintf(fout, " %d %d", vpin->xLoc(), vpin->yLoc());
        if (vpin->orient() != -1)
            fprintf(fout, " %s", orientStr(vpin->orient()));
    }
    fprintf(fout, "\n");
}

// regularWiring
if (net->defiNet::numWires()) {
   for (i = 0; i < net->defiNet::numWires(); i++) {
      newLayer = 0;
      wire = net->defiNet::wire(i);
      fprintf(fout, "\n  + %s ", wire->wireType());
      count = 0;
      for (j = 0; j < wire->defiWire::numPaths(); j++) {
         p = wire->defiWire::path(j);
```

```
p->initTraverse();
while ((path = (int)p->defiPath::next()) != DEFIPATH_DONE) {
   count++;
   // Don't want the line to be too long
   if (count >= 5) {
       fprintf(fout, "\n");
       count = 0;
   }
   switch (path) {
     case DEFIPATH_LAYER:
          if (newLayer == 0) {
              fprintf(fout, "%s ", p->defiPath::getLayer());
              newLayer = 1;
          } else
              fprintf(fout, "NEW %s ", p->defiPath::getLayer());
          break;
     case DEFIPATH_VIA:
          fprintf(fout, "%s ", p->defiPath::getVia());
          break;
     case DEFIPATH_VIAROTATION:
          fprintf(fout, "%s ",
                  orientStr(p->defiPath::getViaRotation()));
          break;
     case DEFIPATH_WIDTH:
          fprintf(fout, "%d ", p->defiPath::getWidth());
          break;
     case DEFIPATH_POINT:
          p->defiPath::getPoint(&x, &y);
          fprintf(fout, "( %d %d ) ", x, y);
          break;
     case DEFIPATH_FLUSHPOINT:
          p->defiPath::getFlushPoint(&x, &y, &z);
          fprintf(fout, "( %d %d %d ) ", x, y, z);
          break;
     case DEFIPATH_TAPER:
          fprintf(fout, "TAPER ");
          break;
     case DEFIPATH_TAPERRULE:
          fprintf(fout, "TAPERRULE %s ",p->defiPath::getTaperRule());
          break;
     case DEFIPATH_STYLE:
```

```
                    fprintf(fout, "STYLE %d ",p->defiPath::getStyle());
                    break;
                }
            }
        }
        fprintf(fout, "\n");
        count = 0;
    }
}

// SHIELDNET
if (net->defiNet::numShieldNets()) {
    for (i = 0; i < net->defiNet::numShieldNets(); i++)
        fprintf(fout, "\n  + SHIELDNET %s", net->defiNet::shieldNet(i));
}

if (net->defiNet::hasSubnets()) {
    for (i = 0; i < net->defiNet::numSubnets(); i++) {
        s = net->defiNet::subnet(i);
        fprintf(fout, "\n");

        if (s->defiSubnet::numConnections()) {
            if (s->defiSubnet::pinIsMustJoin(0))
                fprintf(fout, "- MUSTJOIN ");
            else
                fprintf(fout, "  + SUBNET %s ", s->defiSubnet::name());
            for (j = 0; j < s->defiSubnet::numConnections(); j++)
                fprintf(fout, " ( %s %s )\n", s->defiSubnet::instance(j),
                        s->defiSubnet::pin(j));

            // regularWiring
            if (s->defiSubnet::numWires()) {
                for (k = 0; k < s->defiSubnet::numWires(); k++) {
                    newLayer = 0;
                    wire = s->defiSubnet::wire(k);
                    fprintf(fout, "  %s ", wire->wireType());
                    count = 0;
                    for (j = 0; j < wire->defiWire::numPaths(); j++) {
                        p = wire->defiWire::path(j);
                        p->initTraverse();
                        while ((path = (int)p->defiPath::next()) != DEFIPATH_DONE) {
```

```
count++;
// Don't want the line to be too long
if (count >= 5) {
    fprintf(fout, "\n");
    count = 0;
}
switch (path) {
  case DEFIPATH_LAYER:
       if (newLayer == 0) {
           fprintf(fout, "%s ", p->defiPath::getLayer());
           newLayer = 1;
       } else
           fprintf(fout, "NEW %s ",
                   p->defiPath::getLayer());
       break;
  case DEFIPATH_VIA:
       fprintf(fout, "%s ", p->defiPath::getVia());
       break;
  case DEFIPATH_VIAROTATION:
       fprintf(fout, "%s ",
               p->defiPath::getViaRotationStr());
       break;
  case DEFIPATH_WIDTH:
       fprintf(fout, "%d ", p->defiPath::getWidth());
       break;
  case DEFIPATH_POINT:
       p->defiPath::getPoint(&x, &y);
       fprintf(fout, "( %d %d ) ", x, y);
       break;
  case DEFIPATH_FLUSHPOINT:
       p->defiPath::getFlushPoint(&x, &y, &z);
       fprintf(fout, "( %d %d %d ) ", x, y, z);
       break;
  case DEFIPATH_TAPER:
       fprintf(fout, "TAPER ");
       break;
  case DEFIPATH_TAPERRULE:
       fprintf(fout, "TAPERRULE  %s ",
               p->defiPath::getTaperRule());
       break;
  case DEFIPATH_STYLE:
```

```
                              fprintf(fout, "STYLE  %d ",
                                      p->defiPath::getStyle());
                              break;
                       }
                    }
                 }
              }
           }
         }
      }
   }

   if (net->defiNet::numProps()) {
     for (i = 0; i < net->defiNet::numProps(); i++) {
         fprintf(fout, "  + PROPERTY %s ", net->defiNet::propName(i));
         switch (net->defiNet::propType(i)) {
            case 'R': fprintf(fout, "%g REAL ", net->defiNet::propNumber(i));
                      break;
            case 'I': fprintf(fout, "%g INTEGER ", net->defiNet::propNumber(i));
                      break;
            case 'S': fprintf(fout, "%s STRING ", net->defiNet::propValue(i));
                      break;
           case 'Q': fprintf(fout, "%s QUOTESTRING ", net->defiNet::propValue(i));
                      break;
            case 'N': fprintf(fout, "%g NUMBER ", net->defiNet::propNumber(i));
                      break;
         }
         fprintf(fout, "\n");
     }
   }

   if (net->defiNet::hasWeight())
     fprintf(fout, "+ WEIGHT %d ", net->defiNet::weight());
   if (net->defiNet::hasCap())
     fprintf(fout, "+ ESTCAP %g ", net->defiNet::cap());
   if (net->defiNet::hasSource())
     fprintf(fout, "+ SOURCE %s ", net->defiNet::source());
   if (net->defiNet::hasFixedbump())
     fprintf(fout, "+ FIXEDBUMP ");
   if (net->defiNet::hasFrequency())
     fprintf(fout, "+ FREQUENCY %g ", net->defiNet::frequency());
```

```
  if (net->defiNet::hasPattern())
    fprintf(fout, "+ PATTERN %s ", net->defiNet::pattern());
  if (net->defiNet::hasOriginal())
    fprintf(fout, "+ ORIGINAL %s ", net->defiNet::original());
  if (net->defiNet::hasUse())
    fprintf(fout, "+ USE %s ", net->defiNet::use());

  fprintf (fout, ";\n");
  --numObjs;
  if (numObjs <= 0)
      fprintf(fout, "END NETS\n");
  return 0;
}


int snetpath(defrCallbackType_e c, defiNet* ppath, defiUserData ud) {
  int          i, j, x, y, z, count, newLayer;
  char*        layerName;
  double       dist, left, right;
  defiPath*    p;
  defiSubnet   *s;
  int          path;
  defiShield* shield;
  defiWire*    wire;
  int          numX, numY, stepX, stepY;

  if (c != defrSNetPartialPathCbkType)
      return 1;
  if ((long)ud != userData) dataError();

  fprintf (fout, "SPECIALNET partial data\n");

  fprintf(fout, "- %s ", ppath->defiNet::name());

  count = 0;
  // compName & pinName
  for (i = 0; i < ppath->defiNet::numConnections(); i++) {
      // set the limit of only 5 items print out in one line
      count++;
      if (count >= 5) {
          fprintf(fout, "\n");
```

```
        count = 0;
     }
     fprintf (fout, "( %s %s ) ", ppath->defiNet::instance(i),
              ppath->defiNet::pin(i));
     if (ppath->defiNet::pinIsSynthesized(i))
         fprintf(fout, "+ SYNTHESIZED ");
 }

 // specialWiring
 // POLYGON
 if (ppath->defiNet::numPolygons()) {
    struct defiPoints points;
   for (i = 0; i < ppath->defiNet::numPolygons(); i++) {
     fprintf(fout, "\n  + POLYGON %s ", ppath->polygonName(i));
     points = ppath->getPolygon(i);
     for (j = 0; j < points.numPoints; j++)
       fprintf(fout, "%d %d ", points.x[j], points.y[j]);
   }
 }
 // RECT
 if (ppath->defiNet::numRectangles()) {
    for (i = 0; i < ppath->defiNet::numRectangles(); i++) {
      fprintf(fout, "\n  + RECT %s %d %d %d %d", ppath->defiNet::rectName(i),
              ppath->defiNet::xl(i), ppath->defiNet::yl(i),
              ppath->defiNet::xh(i), ppath->defiNet::yh(i));
    }
 }

 // COVER, FIXED, ROUTED or SHIELD
 if (ppath->defiNet::numWires()) {
    newLayer = 0;
    for (i = 0; i < ppath->defiNet::numWires(); i++) {
       newLayer = 0;
       wire = ppath->defiNet::wire(i);
       fprintf(fout, "\n  + %s ", wire->wireType());
       if (strcmp (wire->wireType(), "SHIELD") == 0)
          fprintf(fout, "%s ", wire->wireShieldNetName());
       for (j = 0; j < wire->defiWire::numPaths(); j++) {
          p = wire->defiWire::path(j);
          p->initTraverse();
          while ((path = (int)p->defiPath::next()) != DEFIPATH_DONE) {
```

```
count++;
// Don't want the line to be too long
if (count >= 5) {
    fprintf(fout, "\n");
    count = 0;
}
switch (path) {
  case DEFIPATH_LAYER:
       if (newLayer == 0) {
           fprintf(fout, "%s ", p->defiPath::getLayer());
           newLayer = 1;
       } else
           fprintf(fout, "NEW %s ", p->defiPath::getLayer());
       break;
  case DEFIPATH_VIA:
       fprintf(fout, "%s ", p->defiPath::getVia());
       break;
  case DEFIPATH_VIAROTATION:
       fprintf(fout, "%s ",
               orientStr(p->defiPath::getViaRotation()));
       break;
  case DEFIPATH_VIADATA:
       p->defiPath::getViaData(&numX, &numY, &stepX, &stepY);
       fprintf(fout, "DO %d BY %d STEP %d %d ", numX, numY,
               stepX, stepY);
       break;
  case DEFIPATH_WIDTH:
       fprintf(fout, "%d ", p->defiPath::getWidth());
       break;
  case DEFIPATH_POINT:
       p->defiPath::getPoint(&x, &y);
       fprintf(fout, "( %d %d ) ", x, y);
       break;
  case DEFIPATH_FLUSHPOINT:
       p->defiPath::getFlushPoint(&x, &y, &z);
       fprintf(fout, "( %d %d %d ) ", x, y, z);
       break;
  case DEFIPATH_TAPER:
       fprintf(fout, "TAPER ");
       break;
  case DEFIPATH_SHAPE:
```

```
                    fprintf(fout, "+ SHAPE %s ", p->defiPath::getShape());
                    break;
               case DEFIPATH_STYLE:
                    fprintf(fout, "+ STYLE %d ", p->defiPath::getStyle());
                    break;
             }
           }
        }
        fprintf(fout, "\n");
        count = 0;
     }
  }

  if (ppath->defiNet::hasSubnets()) {
    for (i = 0; i < ppath->defiNet::numSubnets(); i++) {
      s = ppath->defiNet::subnet(i);
      if (s->defiSubnet::numConnections()) {
          if (s->defiSubnet::pinIsMustJoin(0))
              fprintf(fout, "- MUSTJOIN ");
          else
              fprintf(fout, "- %s ", s->defiSubnet::name());
          for (j = 0; j < s->defiSubnet::numConnections(); j++) {
              fprintf(fout, " ( %s %s )\n", s->defiSubnet::instance(j),
                      s->defiSubnet::pin(j));
        }
      }

      // regularWiring
      if (s->defiSubnet::numWires()) {
         for (i = 0; i < s->defiSubnet::numWires(); i++) {
            wire = s->defiSubnet::wire(i);
            fprintf(fout, "  + %s ", wire->wireType());
            for (j = 0; j < wire->defiWire::numPaths(); j++) {
              p = wire->defiWire::path(j);
              p->defiPath::print(fout);
            }
         }
      }
    }
  }
```

```
if (ppath->defiNet::numProps()) {
  for (i = 0; i < ppath->defiNet::numProps(); i++) {
      if (ppath->defiNet::propIsString(i))
         fprintf(fout, "  + PROPERTY %s %s ", ppath->defiNet::propName(i),
                 ppath->defiNet::propValue(i));
      if (ppath->defiNet::propIsNumber(i))
         fprintf(fout, "  + PROPERTY %s %g ", ppath->defiNet::propName(i),
                 ppath->defiNet::propNumber(i));
      switch (ppath->defiNet::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");
  }
}

// SHIELD
count = 0;
// testing the SHIELD for 5.3, obsolete in 5.4
if (ppath->defiNet::numShields()) {
  for (i = 0; i < ppath->defiNet::numShields(); i++) {
     shield = ppath->defiNet::shield(i);
     fprintf(fout, "\n  + SHIELD %s ", shield->defiShield::shieldName());
     newLayer = 0;
     for (j = 0; j < shield->defiShield::numPaths(); j++) {
        p = shield->defiShield::path(j);
        p->initTraverse();
        while ((path = (int)p->defiPath::next()) != DEFIPATH_DONE) {
           count++;
           // Don't want the line to be too long
           if (count >= 5) {
               fprintf(fout, "\n");
               count = 0;
```

```
    }
    switch (path) {
      case DEFIPATH_LAYER:
           if (newLayer == 0) {
               fprintf(fout, "%s ", p->defiPath::getLayer());
               newLayer = 1;
           } else
               fprintf(fout, "NEW %s ", p->defiPath::getLayer());
           break;
      case DEFIPATH_VIA:
           fprintf(fout, "%s ", p->defiPath::getVia());
           break;
      case DEFIPATH_VIAROTATION:
           if (newLayer)
              fprintf(fout, "%s ",
                      orientStr(p->defiPath::getViaRotation()));
           else
              fprintf(fout, "Str %s ",
                      p->defiPath::getViaRotationStr());
           break;
      case DEFIPATH_WIDTH:
           fprintf(fout, "%d ", p->defiPath::getWidth());
           break;
      case DEFIPATH_POINT:
           p->defiPath::getPoint(&x, &y);
           fprintf(fout, "( %d %d ) ", x, y);
           break;
      case DEFIPATH_FLUSHPOINT:
           p->defiPath::getFlushPoint(&x, &y, &z);
           fprintf(fout, "( %d %d %d ) ", x, y, z);
           break;
      case DEFIPATH_TAPER:
           fprintf(fout, "TAPER ");
           break;
      case DEFIPATH_SHAPE:
           fprintf(fout, "+ SHAPE %s ", p->defiPath::getShape());
           break;
      case DEFIPATH_STYLE:
           fprintf(fout, "+ STYLE %d ", p->defiPath::getStyle());
    }
}
```

```
      }
    }
  }

  // layerName width
  if (ppath->defiNet::hasWidthRules()) {
    for (i = 0; i < ppath->defiNet::numWidthRules(); i++) {
        ppath->defiNet::widthRule(i, &layerName, &dist);
        fprintf (fout, "\n  + WIDTH %s %g ", layerName, dist);
    }
  }

  // layerName spacing
  if (ppath->defiNet::hasSpacingRules()) {
    for (i = 0; i < ppath->defiNet::numSpacingRules(); i++) {
        ppath->defiNet::spacingRule(i, &layerName, &dist, &left, &right);
        if (left == right)
            fprintf (fout, "\n  + SPACING %s %g ", layerName, dist);
        else
            fprintf (fout, "\n  + SPACING %s %g RANGE %g %g ",
                     layerName, dist, left, right);
    }
  }

  if (ppath->defiNet::hasFixedbump())
    fprintf(fout, "\n  + FIXEDBUMP ");
  if (ppath->defiNet::hasFrequency())
    fprintf(fout, "\n  + FREQUENCY %g ", ppath->defiNet::frequency());
  if (ppath->defiNet::hasVoltage())
    fprintf(fout, "\n  + VOLTAGE %g ", ppath->defiNet::voltage());
  if (ppath->defiNet::hasWeight())
    fprintf(fout, "\n  + WEIGHT %d ", ppath->defiNet::weight());
  if (ppath->defiNet::hasCap())
    fprintf(fout, "\n  + ESTCAP %g ", ppath->defiNet::cap());
  if (ppath->defiNet::hasSource())
    fprintf(fout, "\n  + SOURCE %s ", ppath->defiNet::source());
  if (ppath->defiNet::hasPattern())
    fprintf(fout, "\n  + PATTERN %s ", ppath->defiNet::pattern());
  if (ppath->defiNet::hasOriginal())
    fprintf(fout, "\n  + ORIGINAL %s ", ppath->defiNet::original());
  if (ppath->defiNet::hasUse())
```

```
    fprintf(fout, "\n  + USE %s ", ppath->defiNet::use());

  fprintf(fout, "\n");

  return 0;
}


int snetwire(defrCallbackType_e c, defiNet* ppath, defiUserData ud) {
  int         i, j, x, y, z, count = 0, newLayer;
  defiPath*   p;
  int         path;
  defiWire*   wire;
  defiShield* shield;
  int         numX, numY, stepX, stepY;

  if (c != defrSNetWireCbkType)
      return 1;
  if ((long)ud != userData) dataError();

  fprintf (fout, "SPECIALNET wire data\n");

  fprintf(fout, "- %s ", ppath->defiNet::name());

  // specialWiring
  if (ppath->defiNet::numWires()) {
     newLayer = 0;
     for (i = 0; i < ppath->defiNet::numWires(); i++) {
        newLayer = 0;
        wire = ppath->defiNet::wire(i);
        fprintf(fout, "\n  + %s ", wire->wireType());
        if (strcmp (wire->wireType(), "SHIELD") == 0)
           fprintf(fout, "%s ", wire->wireShieldNetName());
        for (j = 0; j < wire->defiWire::numPaths(); j++) {
           p = wire->defiWire::path(j);
           p->initTraverse();
           while ((path = (int)p->defiPath::next()) != DEFIPATH_DONE) {
              count++;
              // Don't want the line to be too long
              if (count >= 5) {
                  fprintf(fout, "\n");
```

```
    count = 0;
  }
  switch (path) {
    case DEFIPATH_LAYER:
         if (newLayer == 0) {
             fprintf(fout, "%s ", p->defiPath::getLayer());
             newLayer = 1;
         } else
             fprintf(fout, "NEW %s ", p->defiPath::getLayer());
         break;
    case DEFIPATH_VIA:
         fprintf(fout, "%s ", p->defiPath::getVia());
         break;
    case DEFIPATH_VIAROTATION:
         fprintf(fout, "%s ",
                 orientStr(p->defiPath::getViaRotation()));
         break;
    case DEFIPATH_VIADATA:
         p->defiPath::getViaData(&numX, &numY, &stepX, &stepY);
         fprintf(fout, "DO %d BY %d STEP %d %d ", numX, numY,
                 stepX, stepY);
         break;
    case DEFIPATH_WIDTH:
         fprintf(fout, "%d ", p->defiPath::getWidth());
         break;
    case DEFIPATH_POINT:
         p->defiPath::getPoint(&x, &y);
         fprintf(fout, "( %d %d ) ", x, y);
         break;
    case DEFIPATH_FLUSHPOINT:
         p->defiPath::getFlushPoint(&x, &y, &z);
         fprintf(fout, "( %d %d %d ) ", x, y, z);
         break;
    case DEFIPATH_TAPER:
         fprintf(fout, "TAPER ");
         break;
    case DEFIPATH_SHAPE:
         fprintf(fout, "+ SHAPE %s ", p->defiPath::getShape());
         break;
    case DEFIPATH_STYLE:
         fprintf(fout, "+ STYLE %d ", p->defiPath::getStyle());
```

```
                     break;
                  }
               }
            }
            fprintf(fout, "\n");
            count = 0;
         }
      } else if (ppath->defiNet::numShields()) {
        for (i = 0; i < ppath->defiNet::numShields(); i++) {
           shield = ppath->defiNet::shield(i);
           fprintf(fout, "\n  + SHIELD %s ", shield->defiShield::shieldName());
           newLayer = 0;
           for (j = 0; j < shield->defiShield::numPaths(); j++) {
              p = shield->defiShield::path(j);
              p->initTraverse();
              while ((path = (int)p->defiPath::next()) != DEFIPATH_DONE) {
                 count++;
                 // Don't want the line to be too long
                 if (count >= 5) {
                     fprintf(fout, "\n");
                     count = 0;
                 }
                 switch (path) {
                   case DEFIPATH_LAYER:
                        if (newLayer == 0) {
                            fprintf(fout, "%s ", p->defiPath::getLayer());
                            newLayer = 1;
                        } else
                            fprintf(fout, "NEW %s ", p->defiPath::getLayer());
                        break;
                   case DEFIPATH_VIA:
                        fprintf(fout, "%s ", p->defiPath::getVia());
                        break;
                   case DEFIPATH_VIAROTATION:
                        fprintf(fout, "%s ",
                                orientStr(p->defiPath::getViaRotation()));
                        break;
                   case DEFIPATH_WIDTH:
                        fprintf(fout, "%d ", p->defiPath::getWidth());
                        break;
                   case DEFIPATH_POINT:
```

```
                    p->defiPath::getPoint(&x, &y);
                    fprintf(fout, "( %d %d ) ", x, y);
                    break;
               case DEFIPATH_FLUSHPOINT:
                    p->defiPath::getFlushPoint(&x, &y, &z);
                    fprintf(fout, "( %d %d %d ) ", x, y, z);
                    break;
               case DEFIPATH_TAPER:
                    fprintf(fout, "TAPER ");
                    break;
               case DEFIPATH_SHAPE:
                    fprintf(fout, "+ SHAPE %s ", p->defiPath::getShape());
                    break;
               case DEFIPATH_STYLE:
                    fprintf(fout, "+ STYLE %d ", p->defiPath::getStyle());
                    break;
             }
          }
       }
    }
  }

  fprintf(fout, "\n");

  return 0;
}

int snetf(defrCallbackType_e c, defiNet* net, defiUserData ud) {
  // For net and special net.
  int         i, j, x, y, z, count, newLayer;
  char*       layerName;
  double      dist, left, right;
  defiPath*   p;
  defiSubnet  *s;
  int         path;
  defiShield* shield;
  defiWire*   wire;
  int         numX, numY, stepX, stepY;

  checkType(c);
  if ((long)ud != userData) dataError();
```

```
if (c != defrSNetCbkType)
    fprintf(fout, "BOGUS NET TYPE  ");

count = 0;
// compName & pinName
for (i = 0; i < net->defiNet::numConnections(); i++) {
    // set the limit of only 5 items print out in one line
    count++;
    if (count >= 5) {
        fprintf(fout, "\n");
        count = 0;
    }
    fprintf (fout, "( %s %s ) ", net->defiNet::instance(i),
             net->defiNet::pin(i));
    if (net->defiNet::pinIsSynthesized(i))
        fprintf(fout, "+ SYNTHESIZED ");
}

// specialWiring
if (net->defiNet::numWires()) {
   newLayer = 0;
   for (i = 0; i < net->defiNet::numWires(); i++) {
      newLayer = 0;
      wire = net->defiNet::wire(i);
      fprintf(fout, "\n  + %s ", wire->wireType());
      if (strcmp (wire->wireType(), "SHIELD") == 0)
         fprintf(fout, "%s ", wire->wireShieldNetName());
      for (j = 0; j < wire->defiWire::numPaths(); j++) {
         p = wire->defiWire::path(j);
         p->initTraverse();
         while ((path = (int)p->defiPath::next()) != DEFIPATH_DONE) {
            count++;
            // Don't want the line to be too long
            if (count >= 5) {
                fprintf(fout, "\n");
                count = 0;
            }
            switch (path) {
              case DEFIPATH_LAYER:
                   if (newLayer == 0) {
                       fprintf(fout, "%s ", p->defiPath::getLayer());
```

```
                newLayer = 1;
            } else
                fprintf(fout, "NEW %s ", p->defiPath::getLayer());
            break;
        case DEFIPATH_VIA:
            fprintf(fout, "%s ", p->defiPath::getVia());
            break;
        case DEFIPATH_VIAROTATION:
            fprintf(fout, "%s ",
                    orientStr(p->defiPath::getViaRotation()));
            break;
        case DEFIPATH_VIADATA:
            p->defiPath::getViaData(&numX, &numY, &stepX, &stepY);
            fprintf(fout, "DO %d BY %d STEP %d %d ", numX, numY,
                    stepX, stepY);
            break;
        case DEFIPATH_WIDTH:
            fprintf(fout, "%d ", p->defiPath::getWidth());
            break;
        case DEFIPATH_POINT:
            p->defiPath::getPoint(&x, &y);
            fprintf(fout, "( %d %d ) ", x, y);
            break;
        case DEFIPATH_FLUSHPOINT:
            p->defiPath::getFlushPoint(&x, &y, &z);
            fprintf(fout, "( %d %d %d ) ", x, y, z);
            break;
        case DEFIPATH_TAPER:
            fprintf(fout, "TAPER ");
            break;
        case DEFIPATH_SHAPE:
            fprintf(fout, "+ SHAPE %s ", p->defiPath::getShape());
            break;
        case DEFIPATH_STYLE:
            fprintf(fout, "+ STYLE %d ", p->defiPath::getStyle());
            break;
      }
   }
}
fprintf(fout, "\n");
count = 0;
```

```
    }
  }
  // POLYGON
  if (net->defiNet::numPolygons()) {
    struct defiPoints points;
    for (i = 0; i < net->defiNet::numPolygons(); i++) {
      fprintf(fout, "\n  + POLYGON %s ", net->polygonName(i));
      points = net->getPolygon(i);
      for (j = 0; j < points.numPoints; j++)
        fprintf(fout, "%d %d ", points.x[j], points.y[j]);
    }
  }
  // RECT
  if (net->defiNet::numRectangles()) {
     for (i = 0; i < net->defiNet::numRectangles(); i++) {
       fprintf(fout, "\n  + RECT %s %d %d %d %d", net->defiNet::rectName(i),
               net->defiNet::xl(i), net->defiNet::yl(i), net->defiNet::xh(i),
               net->defiNet::yh(i));
     }
  }

  if (net->defiNet::hasSubnets()) {
    for (i = 0; i < net->defiNet::numSubnets(); i++) {
      s = net->defiNet::subnet(i);
      if (s->defiSubnet::numConnections()) {
          if (s->defiSubnet::pinIsMustJoin(0))
              fprintf(fout, "- MUSTJOIN ");
          else
              fprintf(fout, "- %s ", s->defiSubnet::name());
          for (j = 0; j < s->defiSubnet::numConnections(); j++) {
              fprintf(fout, " ( %s %s )\n", s->defiSubnet::instance(j),
                      s->defiSubnet::pin(j));
        }
      }

      // regularWiring
      if (s->defiSubnet::numWires()) {
         for (i = 0; i < s->defiSubnet::numWires(); i++) {
            wire = s->defiSubnet::wire(i);
            fprintf(fout, "  + %s ", wire->wireType());
            for (j = 0; j < wire->defiWire::numPaths(); j++) {
```

```
              p = wire->defiWire::path(j);
              p->defiPath::print(fout);
            }
         }
      }
    }
  }

  if (net->defiNet::numProps()) {
    for (i = 0; i < net->defiNet::numProps(); i++) {
        if (net->defiNet::propIsString(i))
           fprintf(fout, "  + PROPERTY %s %s ", net->defiNet::propName(i),
                   net->defiNet::propValue(i));
        if (net->defiNet::propIsNumber(i))
           fprintf(fout, "  + PROPERTY %s %g ", net->defiNet::propName(i),
                   net->defiNet::propNumber(i));
        switch (net->defiNet::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");
    }
  }

  // SHIELD
  count = 0;
  // testing the SHIELD for 5.3, obsolete in 5.4
  if (net->defiNet::numShields()) {
    for (i = 0; i < net->defiNet::numShields(); i++) {
       shield = net->defiNet::shield(i);
       fprintf(fout, "\n  + SHIELD %s ", shield->defiShield::shieldName());
       newLayer = 0;
       for (j = 0; j < shield->defiShield::numPaths(); j++) {
```

```
p = shield->defiShield::path(j);
p->initTraverse();
while ((path = (int)p->defiPath::next()) != DEFIPATH_DONE) {
   count++;
   // Don't want the line to be too long
   if (count >= 5) {
       fprintf(fout, "\n");
       count = 0;
   }
   switch (path) {
     case DEFIPATH_LAYER:
          if (newLayer == 0) {
              fprintf(fout, "%s ", p->defiPath::getLayer());
              newLayer = 1;
          } else
              fprintf(fout, "NEW %s ", p->defiPath::getLayer());
          break;
     case DEFIPATH_VIA:
          fprintf(fout, "%s ", p->defiPath::getVia());
          break;
     case DEFIPATH_VIAROTATION:
          fprintf(fout, "%s ",
                  orientStr(p->defiPath::getViaRotation()));
          break;
     case DEFIPATH_WIDTH:
          fprintf(fout, "%d ", p->defiPath::getWidth());
          break;
     case DEFIPATH_POINT:
          p->defiPath::getPoint(&x, &y);
          fprintf(fout, "( %d %d ) ", x, y);
          break;
     case DEFIPATH_FLUSHPOINT:
          p->defiPath::getFlushPoint(&x, &y, &z);
          fprintf(fout, "( %d %d %d ) ", x, y, z);
          break;
     case DEFIPATH_TAPER:
          fprintf(fout, "TAPER ");
          break;
     case DEFIPATH_SHAPE:
          fprintf(fout, "+ SHAPE %s ", p->defiPath::getShape());
          break;
```

```
              case DEFIPATH_STYLE:
                   fprintf(fout, "+ STYLE %d ", p->defiPath::getStyle());
                   break;
            }
         }
      }
   }
 }

 // layerName width
 if (net->defiNet::hasWidthRules()) {
   for (i = 0; i < net->defiNet::numWidthRules(); i++) {
       net->defiNet::widthRule(i, &layerName, &dist);
       fprintf (fout, "\n  + WIDTH %s %g ", layerName, dist);
   }
 }

 // layerName spacing
 if (net->defiNet::hasSpacingRules()) {
   for (i = 0; i < net->defiNet::numSpacingRules(); i++) {
       net->defiNet::spacingRule(i, &layerName, &dist, &left, &right);
       if (left == right)
           fprintf (fout, "\n  + SPACING %s %g ", layerName, dist);
       else
           fprintf (fout, "\n  + SPACING %s %g RANGE %g %g ",
                    layerName, dist, left, right);
   }
 }

 if (net->defiNet::hasFixedbump())
   fprintf(fout, "\n  + FIXEDBUMP ");
 if (net->defiNet::hasFrequency())
   fprintf(fout, "\n  + FREQUENCY %g ", net->defiNet::frequency());
 if (net->defiNet::hasVoltage())
   fprintf(fout, "\n  + VOLTAGE %g ", net->defiNet::voltage());
 if (net->defiNet::hasWeight())
   fprintf(fout, "\n  + WEIGHT %d ", net->defiNet::weight());
 if (net->defiNet::hasCap())
   fprintf(fout, "\n  + ESTCAP %g ", net->defiNet::cap());
 if (net->defiNet::hasSource())
   fprintf(fout, "\n  + SOURCE %s ", net->defiNet::source());
```

```
  if (net->defiNet::hasPattern())
    fprintf(fout, "\n  + PATTERN %s ", net->defiNet::pattern());
  if (net->defiNet::hasOriginal())
    fprintf(fout, "\n  + ORIGINAL %s ", net->defiNet::original());
  if (net->defiNet::hasUse())
    fprintf(fout, "\n  + USE %s ", net->defiNet::use());

  fprintf (fout, ";\n");
  --numObjs;
  if (numObjs <= 0)
      fprintf(fout, "END SPECIALNETS\n");
  return 0;
}


int ndr(defrCallbackType_e c, defiNonDefault* nd, defiUserData ud) {
  // For nondefaultrule
  int i;

  checkType(c);
  if ((long)ud != userData) dataError();
  if (c != defrNonDefaultCbkType)
      fprintf(fout, "BOGUS NONDEFAULTRULE TYPE  ");
  fprintf(fout, "- %s\n", nd->defiNonDefault::name());
  if (nd->defiNonDefault::hasHardspacing())
      fprintf(fout, "   + HARDSPACING\n");
  for (i = 0; i < nd->defiNonDefault::numLayers(); i++) {
    fprintf(fout, "   + LAYER %s", nd->defiNonDefault::layerName(i));
    fprintf(fout, " WIDTH %d", nd->defiNonDefault::layerWidthVal(i));
    if (nd->defiNonDefault::hasLayerDiagWidth(i))
      fprintf(fout, " DIAGWIDTH %d",
              nd->defiNonDefault::layerDiagWidthVal(i));
    if (nd->defiNonDefault::hasLayerSpacing(i))
      fprintf(fout, " SPACING %d", nd->defiNonDefault::layerSpacingVal(i));
    if (nd->defiNonDefault::hasLayerWireExt(i))
      fprintf(fout, " WIREEXT %d", nd->defiNonDefault::layerWireExtVal(i));
    fprintf(fout, "\n");
  }
  for (i = 0; i < nd->defiNonDefault::numVias(); i++)
    fprintf(fout, "   + VIA %s\n", nd->defiNonDefault::viaName(i));
  for (i = 0; i < nd->defiNonDefault::numViaRules(); i++)
```

```
    fprintf(fout, "    + VIARULE %s\n", nd->defiNonDefault::viaRuleName(i));
  for (i = 0; i < nd->defiNonDefault::numMinCuts(); i++)
    fprintf(fout, "    + MINCUTS %s %d\n", nd->defiNonDefault::cutLayerName(i),
            nd->defiNonDefault::numCuts(i));
  for (i = 0; i < nd->defiNonDefault::numProps(); i++) {
    fprintf(fout, "    + PROPERTY %s %s ", nd->defiNonDefault::propName(i),
            nd->defiNonDefault::propValue(i));
    switch (nd->defiNonDefault::propType(i)) {
      case 'R': fprintf(fout, "REAL\n");
                break;
      case 'I': fprintf(fout, "INTEGER\n");
                break;
      case 'S': fprintf(fout, "STRING\n");
                break;
      case 'Q': fprintf(fout, "QUOTESTRING\n");
                break;
      case 'N': fprintf(fout, "NUMBER\n");
                break;
    }
  }
  --numObjs;
  if (numObjs <= 0)
    fprintf(fout, "END NONDEFAULTRULES\n");
  return 0;
}

int tname(defrCallbackType_e c, const char* string, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "TECHNOLOGY %s ;\n", string);
  return 0;
}

int dname(defrCallbackType_e c, const char* string, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "DESIGN %s ;\n", string);

  // Test changing the user data.
  userData = 89;
  defrSetUserData((void*)userData);
```

```
  return 0;
}


char* address(const char* in) {
  return ((char*)in);
}

int cs(defrCallbackType_e c, int num, defiUserData ud) {
  char* name;

  checkType(c);

  if ((long)ud != userData) dataError();

  switch (c) {
  case defrComponentStartCbkType : name = address("COMPONENTS"); break;
  case defrNetStartCbkType : name = address("NETS"); break;
  case defrStartPinsCbkType : name = address("PINS"); break;
  case defrViaStartCbkType : name = address("VIAS"); break;
  case defrRegionStartCbkType : name = address("REGIONS"); break;
  case defrSNetStartCbkType : name = address("SPECIALNETS"); break;
  case defrGroupsStartCbkType : name = address("GROUPS"); break;
  case defrScanchainsStartCbkType : name = address("SCANCHAINS"); break;
  case defrIOTimingsStartCbkType : name = address("IOTIMINGS"); break;
  case defrFPCStartCbkType : name = address("FLOORPLANCONSTRAINTS"); break;
  case defrTimingDisablesStartCbkType : name = address("TIMING DISABLES"); break;
  case defrPartitionsStartCbkType : name = address("PARTITIONS"); break;
  case defrPinPropStartCbkType : name = address("PINPROPERTIES"); break;
  case defrBlockageStartCbkType : name = address("BLOCKAGES"); break;
  case defrSlotStartCbkType : name = address("SLOTS"); break;
  case defrFillStartCbkType : name = address("FILLS"); break;
  case defrNonDefaultStartCbkType : name = address("NONDEFAULTRULES"); break;
  case defrStylesStartCbkType : name = address("STYLES"); break;
  default : name = address("BOGUS"); return 1;
  }
  fprintf(fout, "\n%s %d ;\n", name, num);
  numObjs = num;
  return 0;
}
```

```
int constraintst(defrCallbackType_e c, int num, defiUserData ud) {
  // Handles both constraints and assertions
  checkType(c);
  if ((long)ud != userData) dataError();
  if (c == defrConstraintsStartCbkType)
      fprintf(fout, "\nCONSTRAINTS %d ;\n\n", num);
  else
      fprintf(fout, "\nASSERTIONS %d ;\n\n", num);
  numObjs = num;
  return 0;
}

void operand(defrCallbackType_e c, defiAssertion* a, int ind) {
  int i, first = 1;
  char* netName;
  char* fromInst, * fromPin, * toInst, * toPin;

  if (a->defiAssertion::isSum()) {
      // Sum in operand, recursively call operand
      fprintf(fout, "- SUM ( ");
      a->defiAssertion::unsetSum();
      isSumSet = 1;
      begOperand = 0;
      operand (c, a, ind);
      fprintf(fout, ") ");
  } else {
      // operand
      if (ind >= a->defiAssertion::numItems()) {
          fprintf(fout, "ERROR: when writing out SUM in Constraints.\n");
          return;
       }
      if (begOperand) {
         fprintf(fout, "- ");
         begOperand = 0;
      }
      for (i = ind; i < a->defiAssertion::numItems(); i++) {
          if (a->defiAssertion::isNet(i)) {
              a->defiAssertion::net(i, &netName);
              if (!first)
                  fprintf(fout, ", "); // print , as separator
```

```
              fprintf(fout, "NET %s ", netName);
          } else if (a->defiAssertion::isPath(i)) {
              a->defiAssertion::path(i, &fromInst, &fromPin, &toInst,
                                     &toPin);
              if (!first)
                  fprintf(fout, ", ");
              fprintf(fout, "PATH %s %s %s %s ", fromInst, fromPin, toInst,
                      toPin);
          } else if (isSumSet) {
              // SUM within SUM, reset the flag
              a->defiAssertion::setSum();
              operand(c, a, i);
          }
          first = 0;
      }

  }
}

int constraint(defrCallbackType_e c, defiAssertion* a, defiUserData ud) {
  // Handles both constraints and assertions

  checkType(c);
  if ((long)ud != userData) dataError();
  if (a->defiAssertion::isWiredlogic())
      // Wirelogic
      fprintf(fout, "- WIREDLOGIC %s + MAXDIST %g ;\n",
              a->defiAssertion::netName(), a->defiAssertion::fallMax());
  else {
      // Call the operand function
      isSumSet = 0;    // reset the global variable
      begOperand = 1;
      operand (c, a, 0);
      // Get the Rise and Fall
      if (a->defiAssertion::hasRiseMax())
          fprintf(fout, "+ RISEMAX %g ", a->defiAssertion::riseMax());
      if (a->defiAssertion::hasFallMax())
          fprintf(fout, "+ FALLMAX %g ", a->defiAssertion::fallMax());
      if (a->defiAssertion::hasRiseMin())
          fprintf(fout, "+ RISEMIN %g ", a->defiAssertion::riseMin());
      if (a->defiAssertion::hasFallMin())
```

```
        fprintf(fout, "+ FALLMIN %g ", a->defiAssertion::fallMin());
    fprintf(fout, ";\n");
  }
  --numObjs;
  if (numObjs <= 0) {
    if (c == defrConstraintCbkType)
        fprintf(fout, "END CONSTRAINTS\n");
    else
        fprintf(fout, "END ASSERTIONS\n");
  }
  return 0;
}


int propstart(defrCallbackType_e c, void* dummy, defiUserData ud) {
  checkType(c);
  fprintf(fout, "\nPROPERTYDEFINITIONS\n");
  isProp = 1;

  return 0;
}


int prop(defrCallbackType_e c, defiProp* p, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  if (strcmp(p->defiProp::propType(), "design") == 0)
      fprintf(fout, "DESIGN %s ", p->defiProp::propName());
  else if (strcmp(p->defiProp::propType(), "net") == 0)
      fprintf(fout, "NET %s ", p->defiProp::propName());
  else if (strcmp(p->defiProp::propType(), "component") == 0)
      fprintf(fout, "COMPONENT %s ", p->defiProp::propName());
  else if (strcmp(p->defiProp::propType(), "specialnet") == 0)
      fprintf(fout, "SPECIALNET %s ", p->defiProp::propName());
  else if (strcmp(p->defiProp::propType(), "group") == 0)
      fprintf(fout, "GROUP %s ", p->defiProp::propName());
  else if (strcmp(p->defiProp::propType(), "row") == 0)
      fprintf(fout, "ROW %s ", p->defiProp::propName());
  else if (strcmp(p->defiProp::propType(), "componentpin") == 0)
      fprintf(fout, "COMPONENTPIN %s ", p->defiProp::propName());
  else if (strcmp(p->defiProp::propType(), "region") == 0)
```

```
      fprintf(fout, "REGION %s ", p->defiProp::propName());
  else if (strcmp(p->defiProp::propType(), "nondefaultrule") == 0)
      fprintf(fout, "NONDEFAULTRULE %s ", p->defiProp::propName());
  if (p->defiProp::dataType() == 'I')
      fprintf(fout, "INTEGER ");
  if (p->defiProp::dataType() == 'R')
      fprintf(fout, "REAL ");
  if (p->defiProp::dataType() == 'S')
      fprintf(fout, "STRING ");
  if (p->defiProp::dataType() == 'Q')
      fprintf(fout, "STRING ");
  if (p->defiProp::hasRange()) {
      fprintf(fout, "RANGE %g %g ", p->defiProp::left(),
              p->defiProp::right());
  }
  if (p->defiProp::hasNumber())
      fprintf(fout, "%g ", p->defiProp::number());
  if (p->defiProp::hasString())
      fprintf(fout, "\"%s\" ", p->defiProp::string());
  fprintf(fout, ";\n");

  return 0;
}


int propend(defrCallbackType_e c, void* dummy, defiUserData ud) {
  checkType(c);
  if (isProp) {
      fprintf(fout, "END PROPERTYDEFINITIONS\n\n");
      isProp = 0;
  }

  defrSetCaseSensitivity(1);
  return 0;
}


int hist(defrCallbackType_e c, const char* h, defiUserData ud) {
  checkType(c);
  defrSetCaseSensitivity(0);
  if ((long)ud != userData) dataError();
```

```
  fprintf(fout, "HISTORY %s ;\n", h);
  defrSetCaseSensitivity(1);
  return 0;
}


int an(defrCallbackType_e c, const char* h, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "ARRAY %s ;\n", h);
  return 0;
}


int fn(defrCallbackType_e c, const char* h, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "FLOORPLAN %s ;\n", h);
  return 0;
}


int bbn(defrCallbackType_e c, const char* h, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "BUSBITCHARS \"%s\" ;\n", h);
  return 0;
}


int vers(defrCallbackType_e c, double d, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "VERSION %g ;\n", d);
    curVer = d;

  defrAddAlias ("alias1", "aliasValue1", 1);
  defrAddAlias ("alias2", "aliasValue2", 0);
  defiAlias_itr *aliasStore;
  aliasStore = (defiAlias_itr*)malloc(sizeof(defiAlias_itr*));
  aliasStore->Init();
```

```
  while (aliasStore->defiAlias_itr::Next()) {
     fprintf(fout, "ALIAS %s %s %d ;\n", aliasStore->defiAlias_itr::Key(),
                   aliasStore->defiAlias_itr::Data(),
                   aliasStore->defiAlias_itr::Marked());
  }
  free(aliasStore);
  return 0;
}


int versStr(defrCallbackType_e c, const char* versionName, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "VERSION %s ;\n", versionName);
  return 0;
}


int units(defrCallbackType_e c, double d, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "UNITS DISTANCE MICRONS %g ;\n", d);
  return 0;
}


int casesens(defrCallbackType_e c, int d, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  if (d == 1)
     fprintf(fout, "NAMESCASESENSITIVE ON ;\n", d);
  else
     fprintf(fout, "NAMESCASESENSITIVE OFF ;\n", d);
  return 0;
}


int cls(defrCallbackType_e c, void* cl, defiUserData ud) {
  defiSite* site;  // Site and Canplace and CannotOccupy
  defiBox* box;  // DieArea and
  defiPinCap* pc;
```

```
defiPin* pin;
int i, j;
defiRow* row;
defiTrack* track;
defiGcellGrid* gcg;
defiVia* via;
defiRegion* re;
defiGroup* group;
defiScanchain* sc;
defiIOTiming* iot;
defiFPC* fpc;
defiTimingDisable* td;
defiPartition* part;
defiPinProp* pprop;
defiBlockage* block;
defiSlot* slots;
defiFill* fills;
defiStyles* styles;
int xl, yl, xh, yh;
char *name, *a1, *b1;
char **inst, **inPin, **outPin;
int  *bits;
int  size;
int corner, typ;
const char *itemT;
char dir;
defiPinAntennaModel* aModel;
struct defiPoints points;

checkType(c);
if ((long)ud != userData) dataError();
switch (c) {

case defrSiteCbkType :
       site = (defiSite*)cl;
       fprintf(fout, "SITE %s %g %g %s ", site->defiSite::name(),
               site->defiSite::x_orig(), site->defiSite::y_orig(),
               orientStr(site->defiSite::orient()));
       fprintf(fout, "DO %g BY %g STEP %g %g ;\n",
               site->defiSite::x_num(), site->defiSite::y_num(),
               site->defiSite::x_step(), site->defiSite::y_step());
```

```
        break;
  case defrCanplaceCbkType :
        site = (defiSite*)cl;
        fprintf(fout, "CANPLACE %s %g %g %s ", site->defiSite::name(),
                site->defiSite::x_orig(), site->defiSite::y_orig(),
                orientStr(site->defiSite::orient()));
        fprintf(fout, "DO %g BY %g STEP %g %g ;\n",
                site->defiSite::x_num(), site->defiSite::y_num(),
                site->defiSite::x_step(), site->defiSite::y_step());
        break;
  case defrCannotOccupyCbkType :
        site = (defiSite*)cl;
        fprintf(fout, "CANNOTOCCUPY %s %g %g %s ",
                site->defiSite::name(), site->defiSite::x_orig(),
                site->defiSite::y_orig(), orientStr(site->defiSite::orient()));
        fprintf(fout, "DO %g BY %g STEP %g %g ;\n",
                site->defiSite::x_num(), site->defiSite::y_num(),
                site->defiSite::x_step(), site->defiSite::y_step());
        break;
  case defrDieAreaCbkType :
        box = (defiBox*)cl;
        fprintf(fout, "DIEAREA %d %d %d %d ;\n",
                box->defiBox::xl(), box->defiBox::yl(), box->defiBox::xh(),
                box->defiBox::yh());
        fprintf(fout, "DIEAREA ");
        points = box->defiBox::getPoint();
        for (i = 0; i < points.numPoints; i++)
          fprintf(fout, "%d %d ", points.x[i], points.y[i]);
        fprintf(fout, ";\n");
        break;
  case defrPinCapCbkType :
        pc = (defiPinCap*)cl;
        fprintf(fout, "MINPINS %d WIRECAP %g ;\n",
                pc->defiPinCap::pin(), pc->defiPinCap::cap());
        --numObjs;
        if (numObjs <= 0)
            fprintf(fout, "END DEFAULTCAP\n");
        break;
  case defrPinCbkType :
        pin = (defiPin*)cl;
        fprintf(fout, "- %s + NET %s ", pin->defiPin::pinName(),
```

```
        pin->defiPin::netName());
if (pin->defiPin::hasDirection())
    fprintf(fout, "+ DIRECTION %s ", pin->defiPin::direction());
if (pin->defiPin::hasUse())
    fprintf(fout, "+ USE %s ", pin->defiPin::use());
if (pin->defiPin::hasNetExpr())
    fprintf(fout, "+ NETEXPR \"%s\" ", pin->defiPin::netExpr());
if (pin->defiPin::hasSupplySensitivity())
    fprintf(fout, "+ SUPPLYSENSITIVITY %s ",
            pin->defiPin::supplySensitivity());
if (pin->defiPin::hasGroundSensitivity())
    fprintf(fout, "+ GROUNDSENSITIVITY %s ",
            pin->defiPin::groundSensitivity());
if (pin->defiPin::hasLayer()) {
    struct defiPoints points;
    for (i = 0; i < pin->defiPin::numLayer(); i++) {
       fprintf(fout, "\n  + LAYER %s ", pin->defiPin::layer(i));
       if (pin->defiPin::hasLayerSpacing(i))
         fprintf(fout, "SPACING %d ",
                pin->defiPin::layerSpacing(i));
       if (pin->defiPin::hasLayerDesignRuleWidth(i))
         fprintf(fout, "DESIGNRULEWIDTH %d ",
                pin->defiPin::layerDesignRuleWidth(i));
       pin->defiPin::bounds(i, &xl, &yl, &xh, &yh);
       fprintf(fout, "%d %d %d %d ", xl, yl, xh, yh);
    }
    for (i = 0; i < pin->defiPin::numPolygons(); i++) {
       fprintf(fout, "\n  + POLYGON %s ",
               pin->defiPin::polygonName(i));
       if (pin->defiPin::hasPolygonSpacing(i))
         fprintf(fout, "SPACING %d ",
                pin->defiPin::polygonSpacing(i));
       if (pin->defiPin::hasPolygonDesignRuleWidth(i))
         fprintf(fout, "DESIGNRULEWIDTH %d ",
                pin->defiPin::polygonDesignRuleWidth(i));
       points = pin->defiPin::getPolygon(i);
       for (j = 0; j < points.numPoints; j++)
         fprintf(fout, "%d %d ", points.x[j], points.y[j]);
    }
    for (i = 0; i < pin->defiPin::numVias(); i++) {
       fprintf(fout, "\n  + VIA %s %d %d ", pin->defiPin::viaName(i),
```

```
                pin->defiPin::viaPtX(i), pin->defiPin::viaPtY(i));
        }
    }
    if (pin->defiPin::hasPort()) {
        struct defiPoints points;
        defiPinPort* port;
        for (j = 0; j < pin->defiPin::numPorts(); j++) {
           port = pin->defiPin::pinPort(j);
           fprintf(fout, "\n  + PORT");
           for (i = 0; i < port->defiPinPort::numLayer(); i++) {
              fprintf(fout, "\n     + LAYER %s ",
                      port->defiPinPort::layer(i));
              if (port->defiPinPort::hasLayerSpacing(i))
                fprintf(fout, "SPACING %d ",
                       port->defiPinPort::layerSpacing(i));
              if (port->defiPinPort::hasLayerDesignRuleWidth(i))
                fprintf(fout, "DESIGNRULEWIDTH %d ",
                       port->defiPinPort::layerDesignRuleWidth(i));
              port->defiPinPort::bounds(i, &xl, &yl, &xh, &yh);
              fprintf(fout, "%d %d %d %d ", xl, yl, xh, yh);
           }
           for (i = 0; i < port->defiPinPort::numPolygons(); i++) {
              fprintf(fout, "\n     + POLYGON %s ",
                      port->defiPinPort::polygonName(i));
              if (port->defiPinPort::hasPolygonSpacing(i))
                fprintf(fout, "SPACING %d ",
                       port->defiPinPort::polygonSpacing(i));
              if (port->defiPinPort::hasPolygonDesignRuleWidth(i))
                fprintf(fout, "DESIGNRULEWIDTH %d ",
                       port->defiPinPort::polygonDesignRuleWidth(i));
              points = port->defiPinPort::getPolygon(i);
              for (j = 0; j < points.numPoints; j++)
                fprintf(fout, "%d %d ", points.x[j], points.y[j]);
           }
           for (i = 0; i < port->defiPinPort::numVias(); i++) {
              fprintf(fout, "\n     + VIA %s %g %g",
                      port->defiPinPort::viaName(i),
                      port->defiPinPort::viaPtX(i),
                      port->defiPinPort::viaPtY(i));
           }
           if (port->defiPinPort::hasPlacement()) {
```

```
          if (port->defiPinPort::isPlaced()) {
             fprintf(fout, "\n     + PLACED ");
             fprintf(fout, "( %d %d ) %s ",
                port->defiPinPort::placementX(),
                port->defiPinPort::placementY(),
                orientStr(port->defiPinPort::orient()));
          }
          if (port->defiPinPort::isCover()) {
             fprintf(fout, "\n     + COVER ");
             fprintf(fout, "( %d %d ) %s ",
                port->defiPinPort::placementX(),
                port->defiPinPort::placementY(),
                orientStr(port->defiPinPort::orient()));
          }
          if (port->defiPinPort::isFixed()) {
             fprintf(fout, "\n     + FIXED ");
             fprintf(fout, "( %d %d ) %s ",
                port->defiPinPort::placementX(),
                port->defiPinPort::placementY(),
                orientStr(port->defiPinPort::orient()));
          }
       }
    }
}
if (pin->defiPin::hasPlacement()) {
     if (pin->defiPin::isPlaced()) {
          fprintf(fout, "+ PLACED ");
          fprintf(fout, "( %d %d ) %s ", pin->defiPin::placementX(),
              pin->defiPin::placementY(),
              orientStr(pin->defiPin::orient()));
    }
     if (pin->defiPin::isCover()) {
          fprintf(fout, "+ COVER ");
          fprintf(fout, "( %d %d ) %s ", pin->defiPin::placementX(),
              pin->defiPin::placementY(),
              orientStr(pin->defiPin::orient()));
     }
     if (pin->defiPin::isFixed()) {
          fprintf(fout, "+ FIXED ");
          fprintf(fout, "( %d %d ) %s ", pin->defiPin::placementX(),
              pin->defiPin::placementY(),
```

```
                orientStr(pin->defiPin::orient()));
        }
        if (pin->defiPin::isUnplaced())
            fprintf(fout, "+ UNPLACED ");
    }
    if (pin->defiPin::hasSpecial()) {
        fprintf(fout, "+ SPECIAL ");
    }
    if (pin->hasAPinPartialMetalArea()) {
        for (i = 0; i < pin->defiPin::numAPinPartialMetalArea(); i++) {
           fprintf(fout, "ANTENNAPINPARTIALMETALAREA %d",
                   pin->APinPartialMetalArea(i));
           if (*(pin->APinPartialMetalAreaLayer(i)))
               fprintf(fout, " LAYER %s",
                       pin->APinPartialMetalAreaLayer(i));
           fprintf(fout, "\n");
        }
    }
    if (pin->hasAPinPartialMetalSideArea()) {
        for (i = 0; i < pin->defiPin::numAPinPartialMetalSideArea(); i++) {
           fprintf(fout, "ANTENNAPINPARTIALMETALSIDEAREA %d",
                   pin->APinPartialMetalSideArea(i));
           if (*(pin->APinPartialMetalSideAreaLayer(i)))
               fprintf(fout, " LAYER %s",
                   pin->APinPartialMetalSideAreaLayer(i));
           fprintf(fout, "\n");
        }
    }
    if (pin->hasAPinDiffArea()) {
        for (i = 0; i < pin->defiPin::numAPinDiffArea(); i++) {
           fprintf(fout, "ANTENNAPINDIFFAREA %d", pin->APinDiffArea(i));
           if (*(pin->APinDiffAreaLayer(i)))
               fprintf(fout, " LAYER %s", pin->APinDiffAreaLayer(i));
           fprintf(fout, "\n");
        }
    }
    if (pin->hasAPinPartialCutArea()) {
        for (i = 0; i < pin->defiPin::numAPinPartialCutArea(); i++) {
           fprintf(fout, "ANTENNAPINPARTIALCUTAREA %d",
                   pin->APinPartialCutArea(i));
           if (*(pin->APinPartialCutAreaLayer(i)))
```

```
            fprintf(fout, " LAYER %s", pin->APinPartialCutAreaLayer(i));
        fprintf(fout, "\n");
     }
 }

 for (j = 0; j < pin->numAntennaModel(); j++) {
    aModel = pin->antennaModel(j);

    fprintf(fout, "ANTENNAMODEL %s\n",
            aModel->defiPinAntennaModel::antennaOxide());

    if (aModel->hasAPinGateArea()) {
        for (i = 0; i < aModel->defiPinAntennaModel::numAPinGateArea();
           i++) {
           fprintf(fout, "ANTENNAPINGATEAREA %d",
                   aModel->APinGateArea(i));
           if (aModel->hasAPinGateAreaLayer(i))
               fprintf(fout, " LAYER %s", aModel->APinGateAreaLayer(i));
           fprintf(fout, "\n");
        }
    }
    if (aModel->hasAPinMaxAreaCar()) {
        for (i = 0;
           i < aModel->defiPinAntennaModel::numAPinMaxAreaCar(); i++) {
           fprintf(fout, "ANTENNAPINMAXAREACAR %d",
                   aModel->APinMaxAreaCar(i));
           if (aModel->hasAPinMaxAreaCarLayer(i))
               fprintf(fout,
                   " LAYER %s", aModel->APinMaxAreaCarLayer(i));
           fprintf(fout, "\n");
        }
    }
    if (aModel->hasAPinMaxSideAreaCar()) {
        for (i = 0;
             i < aModel->defiPinAntennaModel::numAPinMaxSideAreaCar();
             i++) {
           fprintf(fout, "ANTENNAPINMAXSIDEAREACAR %d",
                   aModel->APinMaxSideAreaCar(i));
           if (aModel->hasAPinMaxSideAreaCarLayer(i))
               fprintf(fout,
                   " LAYER %s", aModel->APinMaxSideAreaCarLayer(i));
```

```
                fprintf(fout, "\n");
             }
         }
         if (aModel->hasAPinMaxCutCar()) {
             for (i = 0; i < aModel->defiPinAntennaModel::numAPinMaxCutCar();
                i++) {
                fprintf(fout, "ANTENNAPINMAXCUTCAR %d",
                    aModel->APinMaxCutCar(i));
                if (aModel->hasAPinMaxCutCarLayer(i))
                    fprintf(fout, " LAYER %s",
                    aModel->APinMaxCutCarLayer(i));
                fprintf(fout, "\n");
             }
         }
      }
      fprintf(fout, ";\n");
      --numObjs;
      if (numObjs <= 0)
          fprintf(fout, "END PINS\n");
      break;
case defrDefaultCapCbkType :
      i = (long)cl;
      fprintf(fout, "DEFAULTCAP %d\n", i);
      numObjs = i;
      break;
case defrRowCbkType :
      row = (defiRow*)cl;
      fprintf(fout, "ROW %s %s %g %g %s ", row->defiRow::name(),
              row->defiRow::macro(), row->defiRow::x(), row->defiRow::y(),
              orientStr(row->defiRow::orient()));
      if (row->defiRow::hasDo()) {
          fprintf(fout, "DO %g BY %g ",
                  row->defiRow::xNum(), row->defiRow::yNum());
          if (row->defiRow::hasDoStep())
              fprintf(fout, "STEP %g %g ;\n",
                      row->defiRow::xStep(), row->defiRow::yStep());
          else
              fprintf(fout, ";\n");
      } else
         fprintf(fout, ";\n");
      if (row->defiRow::numProps() > 0) {
```

```
            for (i = 0; i < row->defiRow::numProps(); i++) {
                fprintf(fout, "  + PROPERTY %s %s ",
                        row->defiRow::propName(i),
                        row->defiRow::propValue(i));
                switch (row->defiRow::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");
         }
         break;
  case defrTrackCbkType :
         track = (defiTrack*)cl;
         fprintf(fout, "TRACKS %s %g DO %g STEP %g LAYER ",
                 track->defiTrack::macro(), track->defiTrack::x(),
                 track->defiTrack::xNum(), track->defiTrack::xStep());
         for (i = 0; i < track->defiTrack::numLayers(); i++)
            fprintf(fout, "%s ", track->defiTrack::layer(i));
         fprintf(fout, ";\n");
         break;
  case defrGcellGridCbkType :
         gcg = (defiGcellGrid*)cl;
         fprintf(fout, "GCELLGRID %s %d DO %d STEP %g ;\n",
                 gcg->defiGcellGrid::macro(), gcg->defiGcellGrid::x(),
                 gcg->defiGcellGrid::xNum(), gcg->defiGcellGrid::xStep());
         break;
  case defrViaCbkType :
         via = (defiVia*)cl;
         fprintf(fout, "- %s ", via->defiVia::name());
         if (via->defiVia::hasPattern())
             fprintf(fout, "+ PATTERNNAME %s ", via->defiVia::pattern());
         for (i = 0; i < via->defiVia::numLayers(); i++) {
```

```
        via->defiVia::layer(i, &name, &xl, &yl, &xh, &yh);
        fprintf(fout, "+ RECT %s %d %d %d %d \n",
                name, xl, yl, xh, yh);
    }
    // POLYGON
    if (via->defiVia::numPolygons()) {
      struct defiPoints points;
      for (i = 0; i < via->defiVia::numPolygons(); i++) {
        fprintf(fout, "\n  + POLYGON %s ", via->polygonName(i));
        points = via->getPolygon(i);
        for (j = 0; j < points.numPoints; j++)
          fprintf(fout, "%d %d ", points.x[j], points.y[j]);
      }
    }
    fprintf(fout, " ;\n");
    if (via->defiVia::hasViaRule()) {
        char *vrn, *bl, *cl, *tl;
        int xs, ys, xcs, ycs, xbe, ybe, xte, yte;
        int cr, cc, xo, yo, xbo, ybo, xto, yto;
        (void)via->defiVia::viaRule(&vrn, &xs, &ys, &bl, &cl, &tl, &xcs,
                                     &ycs, &xbe, &ybe, &xte, &yte);
        fprintf(fout, "+ VIARULE '%s'\n", vrn);
        fprintf(fout, "  + CUTSIZE %d %d\n", xs, ys);
        fprintf(fout, "  + LAYERS %s %s %s\n", bl, cl, tl);
        fprintf(fout, "  + CUTSPACING %d %d\n", xcs, ycs);
        fprintf(fout, "  + ENCLOSURE %d %d %d %d\n", xbe, ybe, xte, yte);
        if (via->defiVia::hasRowCol()) {
           (void)via->defiVia::rowCol(&cr, &cc);
           fprintf(fout, "  + ROWCOL %d %d\n", cr, cc);
        }
        if (via->defiVia::hasOrigin()) {
           (void)via->defiVia::origin(&xo, &yo);
           fprintf(fout, "  + ORIGIN %d %d\n", xo, yo);
        }
        if (via->defiVia::hasOffset()) {
           (void)via->defiVia::offset(&xbo, &ybo, &xto, &yto);
           fprintf(fout, "  + OFFSET %d %d %d %d\n", xbo, ybo, xto, yto);
        }
        if (via->defiVia::hasCutPattern())
           fprintf(fout, "  + PATTERN '%s'\n", via->defiVia::cutPattern());
    }
```

```
        --numObjs;
        if (numObjs <= 0)
            fprintf(fout, "END VIAS\n");
        break;
  case defrRegionCbkType :
        re = (defiRegion*)cl;
        fprintf(fout, "- %s ", re->defiRegion::name());
        for (i = 0; i < re->defiRegion::numRectangles(); i++)
            fprintf(fout, "%d %d %d %d \n", re->defiRegion::xl(i),
                    re->defiRegion::yl(i), re->defiRegion::xh(i),
                    re->defiRegion::yh(i));
        if (re->defiRegion::hasType())
            fprintf(fout, "+ TYPE %s\n", re->defiRegion::type());
        if (re->defiRegion::numProps()) {
            for (i = 0; i < re->defiRegion::numProps(); i++) {
                fprintf(fout, "+ PROPERTY %s %s ", re->defiRegion::propName(i),
                        re->defiRegion::propValue(i));
                switch (re->defiRegion::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");
        --numObjs;
        if (numObjs <= 0) {
            fprintf(fout, "END REGIONS\n");
        }
        break;
  case defrGroupNameCbkType :
        if ((char*)cl) {
            fprintf(fout, "- %s", (char*)cl);
            break;
```

```
        }
  case defrGroupMemberCbkType :
        if ((char*)cl) {
            fprintf(fout, " %s", (char*)cl);
            break;
        }
  case defrGroupCbkType :
        group = (defiGroup*)cl;
        if (group->defiGroup::hasMaxX() | group->defiGroup::hasMaxY()
            | group->defiGroup::hasPerim()) {
            fprintf(fout, "\n  + SOFT ");
            if (group->defiGroup::hasPerim())
                fprintf(fout, "MAXHALFPERIMETER %d ",
                        group->defiGroup::perim());
            if (group->defiGroup::hasMaxX())
                fprintf(fout, "MAXX %d ", group->defiGroup::maxX());
            if (group->defiGroup::hasMaxY())
                fprintf(fout, "MAXY %d ", group->defiGroup::maxY());
        }
        if (group->defiGroup::hasRegionName())
            fprintf(fout, "\n  + REGION %s ", group->defiGroup::regionName());
        if (group->defiGroup::hasRegionBox()) {
            int *gxl, *gyl, *gxh, *gyh;
            int size;
            group->defiGroup::regionRects(&size, &gxl, &gyl, &gxh, &gyh);
            for (i = 0; i < size; i++)
                fprintf(fout, "REGION %d %d %d %d ", gxl[i], gyl[i],
                        gxh[i], gyh[i]);
        }
        if (group->defiGroup::numProps()) {
            for (i = 0; i < group->defiGroup::numProps(); i++) {
                fprintf(fout, "\n  + PROPERTY %s %s ",
                        group->defiGroup::propName(i),
                        group->defiGroup::propValue(i));
                switch (group->defiGroup::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");
       --numObjs;
       if (numObjs <= 0)
           fprintf(fout, "END GROUPS\n");
       break;
  case defrScanchainCbkType :
       sc = (defiScanchain*)cl;
       fprintf(fout, "- %s\n", sc->defiScanchain::name());
       if (sc->defiScanchain::hasStart()) {
           sc->defiScanchain::start(&a1, &b1);
           fprintf(fout, "  + START %s %s\n", a1, b1);
       }
       if (sc->defiScanchain::hasStop()) {
           sc->defiScanchain::stop(&a1, &b1);
           fprintf(fout, "  + STOP %s %s\n", a1, b1);
       }
       if (sc->defiScanchain::hasCommonInPin() ||
           sc->defiScanchain::hasCommonOutPin()) {
           fprintf(fout, "  + COMMONSCANPINS ");
           if (sc->defiScanchain::hasCommonInPin())
              fprintf(fout, " ( IN %s ) ", sc->defiScanchain::commonInPin());
           if (sc->defiScanchain::hasCommonOutPin())
              fprintf(fout, " ( OUT %s ) ",sc->defiScanchain::commonOutPin());
           fprintf(fout, "\n");
       }
       if (sc->defiScanchain::hasFloating()) {
          sc->defiScanchain::floating(&size, &inst, &inPin, &outPin, &bits);
          if (size > 0)
              fprintf(fout, "  + FLOATING\n");
          for (i = 0; i < size; i++) {
              fprintf(fout, "    %s ", inst[i]);
              if (inPin[i])
                 fprintf(fout, "( IN %s ) ", inPin[i]);
              if (outPin[i])
```

```
                fprintf(fout, "( OUT %s ) ", outPin[i]);
             if (bits[i] != -1)
                fprintf(fout, "( BITS %d ) ", bits[i]);
             fprintf(fout, "\n");
          }
       }

       if (sc->defiScanchain::hasOrdered()) {
          for (i = 0; i < sc->defiScanchain::numOrderedLists(); i++) {
              sc->defiScanchain::ordered(i, &size, &inst, &inPin, &outPin,
                                         &bits);
              if (size > 0)
                  fprintf(fout, "  + ORDERED\n");
              for (j = 0; j < size; j++) {
                  fprintf(fout, "    %s ", inst[j]);
                  if (inPin[j])
                     fprintf(fout, "( IN %s ) ", inPin[j]);
                  if (outPin[j])
                     fprintf(fout, "( OUT %s ) ", outPin[j]);
                  if (bits[j] != -1)
                     fprintf(fout, "( BITS %d ) ", bits[j]);
                  fprintf(fout, "\n");
              }
          }
       }

       if (sc->defiScanchain::hasPartition()) {
          fprintf(fout, "  + PARTITION %s ",
                  sc->defiScanchain::partitionName());
          if (sc->defiScanchain::hasPartitionMaxBits())
            fprintf(fout, "MAXBITS %d ",
                    sc->defiScanchain::partitionMaxBits());
       }
       fprintf(fout, ";\n");
       --numObjs;
       if (numObjs <= 0)
           fprintf(fout, "END SCANCHAINS\n");
       break;
 case defrIOTimingCbkType :
       iot = (defiIOTiming*)cl;
       fprintf(fout, "- ( %s %s )\n", iot->defiIOTiming::inst(),
```

```
                iot->defiIOTiming::pin());
        if (iot->defiIOTiming::hasSlewRise())
            fprintf(fout, "  + RISE SLEWRATE %g %g\n",
                    iot->defiIOTiming::slewRiseMin(),
                    iot->defiIOTiming::slewRiseMax());
        if (iot->defiIOTiming::hasSlewFall())
            fprintf(fout, "  + FALL SLEWRATE %g %g\n",
                    iot->defiIOTiming::slewFallMin(),
                    iot->defiIOTiming::slewFallMax());
        if (iot->defiIOTiming::hasVariableRise())
            fprintf(fout, "  + RISE VARIABLE %g %g\n",
                    iot->defiIOTiming::variableRiseMin(),
                    iot->defiIOTiming::variableRiseMax());
        if (iot->defiIOTiming::hasVariableFall())
            fprintf(fout, "  + FALL VARIABLE %g %g\n",
                    iot->defiIOTiming::variableFallMin(),
                    iot->defiIOTiming::variableFallMax());
        if (iot->defiIOTiming::hasCapacitance())
            fprintf(fout, "  + CAPACITANCE %g\n",
                    iot->defiIOTiming::capacitance());
        if (iot->defiIOTiming::hasDriveCell()) {
            fprintf(fout, "  + DRIVECELL %s ",
                    iot->defiIOTiming::driveCell());
            if (iot->defiIOTiming::hasFrom())
                fprintf(fout, "  FROMPIN %s ",
                        iot->defiIOTiming::from());
            if (iot->defiIOTiming::hasTo())
                fprintf(fout, "  TOPIN %s ",
                        iot->defiIOTiming::to());
            if (iot->defiIOTiming::hasParallel())
                fprintf(fout, "PARALLEL %g",
                        iot->defiIOTiming::parallel());
            fprintf(fout, "\n");
        }
        fprintf(fout, ";\n");
        --numObjs;
        if (numObjs <= 0)
            fprintf(fout, "END IOTIMINGS\n");
        break;
  case defrFPCCbkType :
        fpc = (defiFPC*)cl;
```

```
        fprintf(fout, "- %s ", fpc->defiFPC::name());
        if (fpc->defiFPC::isVertical())
            fprintf(fout, "VERTICAL ");
        if (fpc->defiFPC::isHorizontal())
            fprintf(fout, "HORIZONTAL ");
        if (fpc->defiFPC::hasAlign())
            fprintf(fout, "ALIGN ");
        if (fpc->defiFPC::hasMax())
            fprintf(fout, "%g ", fpc->defiFPC::alignMax());
        if (fpc->defiFPC::hasMin())
            fprintf(fout, "%g ", fpc->defiFPC::alignMin());
        if (fpc->defiFPC::hasEqual())
            fprintf(fout, "%g ", fpc->defiFPC::equal());
        for (i = 0; i < fpc->defiFPC::numParts(); i++) {
            fpc->defiFPC::getPart(i, &corner, &typ, &name);
            if (corner == 'B')
                fprintf(fout, "BOTTOMLEFT ");
            else
                fprintf(fout, "TOPRIGHT ");
            if (typ == 'R')
                fprintf(fout, "ROWS %s ", name);
            else
                fprintf(fout, "COMPS %s ", name);
        }
        fprintf(fout, ";\n");
        --numObjs;
        if (numObjs <= 0)
            fprintf(fout, "END FLOORPLANCONSTRAINTS\n");
        break;
  case defrTimingDisableCbkType :
        td = (defiTimingDisable*)cl;
        if (td->defiTimingDisable::hasFromTo())
            fprintf(fout, "- FROMPIN %s %s ",
                    td->defiTimingDisable::fromInst(),
                    td->defiTimingDisable::fromPin(),
                    td->defiTimingDisable::toInst(),
                    td->defiTimingDisable::toPin());
        if (td->defiTimingDisable::hasThru())
            fprintf(fout, "- THRUPIN %s %s ",
                    td->defiTimingDisable::thruInst(),
                    td->defiTimingDisable::thruPin());
```

```
        if (td->defiTimingDisable::hasMacroFromTo())
            fprintf(fout, "- MACRO %s FROMPIN %s %s ",
                    td->defiTimingDisable::macroName(),
                    td->defiTimingDisable::fromPin(),
                    td->defiTimingDisable::toPin());
        if (td->defiTimingDisable::hasMacroThru())
            fprintf(fout, "- MACRO %s THRUPIN %s %s ",
                    td->defiTimingDisable::macroName(),
                    td->defiTimingDisable::fromPin());
        fprintf(fout, ";\n");
        break;
  case defrPartitionCbkType :
        part = (defiPartition*)cl;
        fprintf(fout, "- %s ", part->defiPartition::name());
        if (part->defiPartition::isSetupRise() |
            part->defiPartition::isSetupFall() |
            part->defiPartition::isHoldRise() |
            part->defiPartition::isHoldFall()) {
            // has turnoff
            fprintf(fout, "TURNOFF ");
            if (part->defiPartition::isSetupRise())
                fprintf(fout, "SETUPRISE ");
            if (part->defiPartition::isSetupFall())
                fprintf(fout, "SETUPFALL ");
            if (part->defiPartition::isHoldRise())
                fprintf(fout, "HOLDRISE ");
            if (part->defiPartition::isHoldFall())
                fprintf(fout, "HOLDFALL ");
        }
        itemT = part->defiPartition::itemType();
        dir = part->defiPartition::direction();
        if (strcmp(itemT, "CLOCK") == 0) {
            if (dir == 'T')     // toclockpin
                fprintf(fout, "+ TOCLOCKPIN %s %s ",
                        part->defiPartition::instName(),
                        part->defiPartition::pinName());
            if (dir == 'F')     // fromclockpin
                fprintf(fout, "+ FROMCLOCKPIN %s %s ",
                        part->defiPartition::instName(),
                        part->defiPartition::pinName());
            if (part->defiPartition::hasMin())
```

```
                fprintf(fout, "MIN %g %g ",
                        part->defiPartition::partitionMin(),
                        part->defiPartition::partitionMax());
            if (part->defiPartition::hasMax())
                fprintf(fout, "MAX %g %g ",
                        part->defiPartition::partitionMin(),
                        part->defiPartition::partitionMax());
            fprintf(fout, "PINS ");
            for (i = 0; i < part->defiPartition::numPins(); i++)
                 fprintf(fout, "%s ", part->defiPartition::pin(i));
        } else if (strcmp(itemT, "IO") == 0) {
            if (dir == 'T')    // toiopin
                fprintf(fout, "+ TOIOPIN %s %s ",
                        part->defiPartition::instName(),
                        part->defiPartition::pinName());
            if (dir == 'F')    // fromiopin
                fprintf(fout, "+ FROMIOPIN %s %s ",
                        part->defiPartition::instName(),
                        part->defiPartition::pinName());
        } else if (strcmp(itemT, "COMP") == 0) {
            if (dir == 'T')    // tocomppin
                fprintf(fout, "+ TOCOMPPIN %s %s ",
                        part->defiPartition::instName(),
                        part->defiPartition::pinName());
            if (dir == 'F')    // fromcomppin
                fprintf(fout, "+ FROMCOMPPIN %s %s ",
                        part->defiPartition::instName(),
                        part->defiPartition::pinName());
        }
        fprintf(fout, ";\n");
        --numObjs;
        if (numObjs <= 0)
            fprintf(fout, "END PARTITIONS\n");
        break;

  case defrPinPropCbkType :
        pprop = (defiPinProp*)cl;
        if (pprop->defiPinProp::isPin())
           fprintf(fout, "- PIN %s ", pprop->defiPinProp::pinName());
        else
           fprintf(fout, "- %s %s ",
```

```
                pprop->defiPinProp::instName(),
                pprop->defiPinProp::pinName());
        fprintf(fout, ";\n");
        if (pprop->defiPinProp::numProps() > 0) {
           for (i = 0; i < pprop->defiPinProp::numProps(); i++) {
               fprintf(fout, "  + PROPERTY %s %s ",
                       pprop->defiPinProp::propName(i),
                       pprop->defiPinProp::propValue(i));
               switch (pprop->defiPinProp::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");
        }
        --numObjs;
        if (numObjs <= 0)
            fprintf(fout, "END PINPROPERTIES\n");
        break;
 case defrBlockageCbkType :
        block = (defiBlockage*)cl;
        if (block->defiBlockage::hasLayer()) {
           fprintf(fout, "- LAYER %s\n", block->defiBlockage::layerName());
           if (block->defiBlockage::hasComponent())
              fprintf(fout, "   + COMPONENT %s\n",
                      block->defiBlockage::layerComponentName());
           if (block->defiBlockage::hasSlots())
              fprintf(fout, "   + SLOTS\n");
           if (block->defiBlockage::hasFills())
              fprintf(fout, "   + FILLS\n");
           if (block->defiBlockage::hasPushdown())
              fprintf(fout, "   + PUSHDOWN\n");
           if (block->defiBlockage::hasExceptpgnet())
```

```
            fprintf(fout, "   + EXCEPTPGNET\n");
         if (block->defiBlockage::hasSpacing())
            fprintf(fout, "   + SPACING %d\n",
                    block->defiBlockage::minSpacing());
         if (block->defiBlockage::hasDesignRuleWidth())
            fprintf(fout, "   + DESIGNRULEWIDTH %d\n",
                    block->defiBlockage::designRuleWidth());
      }
      else if (block->defiBlockage::hasPlacement()) {
         fprintf(fout, "- PLACEMENT\n");
         if (block->defiBlockage::hasSoft())
            fprintf(fout, "   + SOFT\n");
         if (block->defiBlockage::hasPartial())
            fprintf(fout, "   + PARTIAL %g\n",
                    block->defiBlockage::placementMaxDensity());
         if (block->defiBlockage::hasComponent())
            fprintf(fout, "   + COMPONENT %s\n",
                    block->defiBlockage::placementComponentName());
         if (block->defiBlockage::hasPushdown())
            fprintf(fout, "   + PUSHDOWN\n");
      }

      for (i = 0; i < block->defiBlockage::numRectangles(); i++) {
         fprintf(fout, "   RECT %d %d %d %d\n",
                 block->defiBlockage::xl(i), block->defiBlockage::yl(i),
                 block->defiBlockage::xh(i), block->defiBlockage::yh(i));
      }

      for (i = 0; i < block->defiBlockage::numPolygons(); i++) {
         fprintf(fout, "   POLYGON ");
         points = block->getPolygon(i);
         for (j = 0; j < points.numPoints; j++)
            fprintf(fout, "%d %d ", points.x[j], points.y[j]);
         fprintf(fout, "\n");
      }
      fprintf(fout, ";\n");
      --numObjs;
      if (numObjs <= 0)
          fprintf(fout, "END BLOCKAGES\n");
      break;
  case defrSlotCbkType :
```

```
        slots = (defiSlot*)cl;
        if (slots->defiSlot::hasLayer())
           fprintf(fout, "- LAYER %s\n", slots->defiSlot::layerName());

        for (i = 0; i < slots->defiSlot::numRectangles(); i++) {
           fprintf(fout, "   RECT %d %d %d %d\n",
                   slots->defiSlot::xl(i), slots->defiSlot::yl(i),
                   slots->defiSlot::xh(i), slots->defiSlot::yh(i));
        }
        for (i = 0; i < slots->defiSlot::numPolygons(); i++) {
           fprintf(fout, "   POLYGON ");
           points = slots->getPolygon(i);
           for (j = 0; j < points.numPoints; j++)
             fprintf(fout, "%d %d ", points.x[j], points.y[j]);
           fprintf(fout, ";\n");
        }
        fprintf(fout, ";\n");
        --numObjs;
        if (numObjs <= 0)
            fprintf(fout, "END SLOTS\n");
        break;
  case defrFillCbkType :
        fills = (defiFill*)cl;
        if (fills->defiFill::hasLayer()) {
           fprintf(fout, "- LAYER %s", fills->defiFill::layerName());
           if (fills->defiFill::hasLayerOpc())
              fprintf(fout, " + OPC");
           fprintf(fout, "\n");

           for (i = 0; i < fills->defiFill::numRectangles(); i++) {
              fprintf(fout, "   RECT %d %d %d %d\n",
                      fills->defiFill::xl(i), fills->defiFill::yl(i),
                      fills->defiFill::xh(i), fills->defiFill::yh(i));
           }
           for (i = 0; i < fills->defiFill::numPolygons(); i++) {
              fprintf(fout, "   POLYGON ");
              points = fills->getPolygon(i);
              for (j = 0; j < points.numPoints; j++)
                fprintf(fout, "%d %d ", points.x[j], points.y[j]);
              fprintf(fout, ";\n");
           }
```

```
            fprintf(fout, ";\n");
         }
         --numObjs;
         if (fills->defiFill::hasVia()) {
            fprintf(fout, "- VIA %s", fills->defiFill::viaName());
            if (fills->defiFill::hasViaOpc())
               fprintf(fout, " + OPC");
            fprintf(fout, "\n");

            for (i = 0; i < fills->defiFill::numViaPts(); i++) {
               points = fills->getViaPts(i);
               for (j = 0; j < points.numPoints; j++)
                  fprintf(fout, " %d %d", points.x[j], points.y[j]);
               fprintf(fout, ";\n");
            }
            fprintf(fout, ";\n");
         }
         if (numObjs <= 0)
             fprintf(fout, "END FILLS\n");
         break;
  case defrStylesCbkType :
         struct defiPoints points;
         styles = (defiStyles*)cl;
         fprintf(fout, "- STYLE %d ", styles->defiStyles::style());
         points = styles->defiStyles::getPolygon();
         for (j = 0; j < points.numPoints; j++)
            fprintf(fout, "%d %d ", points.x[j], points.y[j]);
         fprintf(fout, ";\n");
         --numObjs;
         if (numObjs <= 0)
             fprintf(fout, "END STYLES\n");
         break;

  default: fprintf(fout, "BOGUS callback to cls.\n"); return 1;
  }
  return 0;
}


int dn(defrCallbackType_e c, const char* h, defiUserData ud) {
  checkType(c);
```

```
  if ((long)ud != userData) dataError();
  fprintf(fout, "DIVIDERCHAR \"%s\" ;\n",h);
  return 0;
}


int ext(defrCallbackType_e t, const char* c, defiUserData ud) {
  char* name;

  checkType(t);
  if ((long)ud != userData) dataError();

  switch (t) {
  case defrNetExtCbkType : name = address("net"); break;
  case defrComponentExtCbkType : name = address("component"); break;
  case defrPinExtCbkType : name = address("pin"); break;
  case defrViaExtCbkType : name = address("via"); break;
  case defrNetConnectionExtCbkType : name = address("net connection"); break;
  case defrGroupExtCbkType : name = address("group"); break;
  case defrScanChainExtCbkType : name = address("scanchain"); break;
  case defrIoTimingsExtCbkType : name = address("io timing"); break;
  case defrPartitionsExtCbkType : name = address("partition"); break;
  default: name = address("BOGUS"); return 1;
  }
  fprintf(fout, "  %s extension %s\n", name, c);
  return 0;
}

int extension(defrCallbackType_e c, const char* extsn, defiUserData ud) {
  checkType(c);
  if ((long)ud != userData) dataError();
  fprintf(fout, "BEGINEXT %s\n", extsn);
  return 0;
}

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) {
  int num = 1734;
  char* inFile[6];
  char* outFile;
  FILE* f;
  int res;
  int noCalls = 0;
  int retStr = 0;
  int numInFile = 0;
  int fileCt = 0;

  strcpy(defaultName, "def.in");
  strcpy(defaultOut, "list");
  inFile[0] = defaultName;
  outFile = defaultOut;
  fout = stdout;
  userData = 0x01020304;

  argc--;
  argv++;
  while (argc--) {

    if (strcmp(*argv, "-d") == 0) {
      argv++;
      argc--;
      sscanf(*argv, "%d", &num);
      defiSetDebug(num, 1);

    } else if (strcmp(*argv, "-nc") == 0) {
```

```
      noCalls = 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 (argv[0][0] != '-') {
      if (numInFile >= 6) {
        fprintf(stderr, "ERROR: too many input files, max = 6.\n");
        return 2;
      }
      inFile[numInFile++] = *argv;
    } else if (strcmp(*argv, "-h") == 0) {
      fprintf(stderr, "Usage: defrw [<defFilename>] [-o <outputFilename>]\n");
      return 2;
    } else if (strcmp(*argv, "-setSNetWireCbk") == 0) {
      setSNetWireCbk = 1;
    } else {
      fprintf(stderr, "ERROR: Illegal command line option: '%s'\n", *argv);
      return 2;
    }

    argv++;
  }

  if (noCalls == 0) {
    defrSetUserData((void*)3);
    defrSetDesignCbk(dname);
    defrSetTechnologyCbk(tname);
    defrSetExtensionCbk(extension);
    defrSetDesignEndCbk(done);
```

```
defrSetPropDefStartCbk(propstart);
defrSetPropCbk(prop);
defrSetPropDefEndCbk(propend);
defrSetNetCbk(netf);
defrSetNetNameCbk(netNamef);
defrSetNetNonDefaultRuleCbk(nondefRulef);
defrSetNetSubnetNameCbk(subnetNamef);
defrSetNetPartialPathCbk(netpath);
defrSetSNetCbk(snetf);
defrSetSNetPartialPathCbk(snetpath);
if (setSNetWireCbk)
  defrSetSNetWireCbk(snetwire);
defrSetComponentCbk(compf);
defrSetAddPathToNet();
defrSetHistoryCbk(hist);
defrSetConstraintCbk(constraint);
defrSetAssertionCbk(constraint);
defrSetArrayNameCbk(an);
defrSetFloorPlanNameCbk(fn);
defrSetDividerCbk(dn);
defrSetBusBitCbk(bbn);
defrSetNonDefaultCbk(ndr);

defrSetAssertionsStartCbk(constraintst);
defrSetConstraintsStartCbk(constraintst);
defrSetComponentStartCbk(cs);
defrSetPinPropStartCbk(cs);
defrSetNetStartCbk(cs);
defrSetStartPinsCbk(cs);
defrSetViaStartCbk(cs);
defrSetRegionStartCbk(cs);
defrSetSNetStartCbk(cs);
defrSetGroupsStartCbk(cs);
defrSetScanchainsStartCbk(cs);
defrSetIOTimingsStartCbk(cs);
defrSetFPCStartCbk(cs);
defrSetTimingDisablesStartCbk(cs);
defrSetPartitionsStartCbk(cs);
defrSetBlockageStartCbk(cs);
defrSetSlotStartCbk(cs);
defrSetFillStartCbk(cs);
```

```
defrSetNonDefaultStartCbk(cs);
defrSetStylesStartCbk(cs);

// All of the extensions point to the same function.
defrSetNetExtCbk(ext);
defrSetComponentExtCbk(ext);
defrSetPinExtCbk(ext);
defrSetViaExtCbk(ext);
defrSetNetConnectionExtCbk(ext);
defrSetGroupExtCbk(ext);
defrSetScanChainExtCbk(ext);
defrSetIoTimingsExtCbk(ext);
defrSetPartitionsExtCbk(ext);

defrSetUnitsCbk(units);
if (!retStr)
   defrSetVersionCbk(vers);
else
   defrSetVersionStrCbk(versStr);
defrSetCaseSensitiveCbk(casesens);

// The following calls are an example of using one function "cls"
// to be the callback for many DIFFERENT types of constructs.
// We have to cast the function type to meet the requirements
// of each different set function.
defrSetSiteCbk((defrSiteCbkFnType)cls);
defrSetCanplaceCbk((defrSiteCbkFnType)cls);
defrSetCannotOccupyCbk((defrSiteCbkFnType)cls);
defrSetDieAreaCbk((defrBoxCbkFnType)cls);
defrSetPinCapCbk((defrPinCapCbkFnType)cls);
defrSetPinCbk((defrPinCbkFnType)cls);
defrSetPinPropCbk((defrPinPropCbkFnType)cls);
defrSetDefaultCapCbk((defrIntegerCbkFnType)cls);
defrSetRowCbk((defrRowCbkFnType)cls);
defrSetTrackCbk((defrTrackCbkFnType)cls);
defrSetGcellGridCbk((defrGcellGridCbkFnType)cls);
defrSetViaCbk((defrViaCbkFnType)cls);
defrSetRegionCbk((defrRegionCbkFnType)cls);
defrSetGroupNameCbk((defrStringCbkFnType)cls);
defrSetGroupMemberCbk((defrStringCbkFnType)cls);
defrSetGroupCbk((defrGroupCbkFnType)cls);
```

```
defrSetScanchainCbk((defrScanchainCbkFnType)cls);
defrSetIOTimingCbk((defrIOTimingCbkFnType)cls);
defrSetFPCCbk((defrFPCCbkFnType)cls);
defrSetTimingDisableCbk((defrTimingDisableCbkFnType)cls);
defrSetPartitionCbk((defrPartitionCbkFnType)cls);
defrSetBlockageCbk((defrBlockageCbkFnType)cls);
defrSetSlotCbk((defrSlotCbkFnType)cls);
defrSetFillCbk((defrFillCbkFnType)cls);
defrSetStylesCbk((defrStylesCbkFnType)cls);

defrSetAssertionsEndCbk(endfunc);
defrSetComponentEndCbk(endfunc);
defrSetConstraintsEndCbk(endfunc);
defrSetNetEndCbk(endfunc);
defrSetFPCEndCbk(endfunc);
defrSetFPCEndCbk(endfunc);
defrSetGroupsEndCbk(endfunc);
defrSetIOTimingsEndCbk(endfunc);
defrSetNetEndCbk(endfunc);
defrSetPartitionsEndCbk(endfunc);
defrSetRegionEndCbk(endfunc);
defrSetSNetEndCbk(endfunc);
defrSetScanchainsEndCbk(endfunc);
defrSetPinEndCbk(endfunc);
defrSetTimingDisablesEndCbk(endfunc);
defrSetViaEndCbk(endfunc);
defrSetPinPropEndCbk(endfunc);
defrSetBlockageEndCbk(endfunc);
defrSetSlotEndCbk(endfunc);
defrSetFillEndCbk(endfunc);
defrSetNonDefaultEndCbk(endfunc);
defrSetStylesEndCbk(endfunc);

defrSetMallocFunction(mallocCB);
defrSetReallocFunction(reallocCB);
defrSetFreeFunction(freeCB);

defrSetLineNumberFunction(lineNumberCB);
defrSetDeltaNumberLines(50);

// Testing to set the number of warnings
```

```
    defrSetAssertionWarnings(3);
    defrSetBlockageWarnings(3);
    defrSetCaseSensitiveWarnings(3);
    defrSetComponentWarnings(3);
    defrSetConstraintWarnings(0);
    defrSetDefaultCapWarnings(3);
    defrSetGcellGridWarnings(3);
    defrSetIOTimingWarnings(3);
    defrSetNetWarnings(3);
    defrSetNonDefaultWarnings(3);
    defrSetPinExtWarnings(3);
    defrSetPinWarnings(3);
    defrSetRegionWarnings(3);
    defrSetRowWarnings(3);
    defrSetScanchainWarnings(3);
    defrSetSNetWarnings(3);
    defrSetStylesWarnings(3);
    defrSetTrackWarnings(3);
    defrSetUnitsWarnings(3);
    defrSetVersionWarnings(3);
    defrSetViaWarnings(3);
  }

  defrInit();

  for (fileCt = 0; fileCt < numInFile; fileCt++) {
    defrReset();
    if ((f = fopen(inFile[fileCt],"r")) == 0) {
      fprintf(stderr,"Couldn't open input file '%s'\n", inFile[fileCt]);
      return(2);
    }
    // Set case sensitive to 0 to start with, in History & PropertyDefinition
    // reset it to 1.
    res = defrRead(f, inFile[fileCt], (void*)userData, 1);

    if (res)
       fprintf(stderr, "Reader returns bad status.\n", inFile[fileCt]);

    (void)defrPrintUnusedCallbacks(fout);
    (void)defrReleaseNResetMemory();
```

```
  }
  fclose(fout);

  return res;
}
```

DEF Writer Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#ifndef WIN32
#   include <unistd.h>
#endif /* not WIN32 */
#include "defwWriter.hpp"

char defaultOut[128];

// Global variables
FILE* fout;

#define CHECK_STATUS(status) \
  if (status) {              \
     defwPrintError(status); \
     return(status);         \
  }

int main(int argc, char** argv) {
  char* outfile;
  int   status;    // return code, if none 0 means error
  int   lineNumber = 0;

  const char** layers;
  const char** foreigns;
  int *foreignX, *foreignY, *foreignOrient;
  const char** foreignOrientStr;
  const char **coorX, **coorY;
  const char **coorValue;
  const char **groupExpr;
  int *xPoints, *yPoints;
```

```
double *xP, *yP;

// assign the default
strcpy(defaultOut, "def.in");
outfile = defaultOut;
fout = stdout;

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: defwrite [-o <filename>] [-help]\n");
      return 1;
   } else {
      fprintf(stderr, "ERROR: Illegal command line option: '%s'\n", *argv);
      return 2;
   }
   argv++;
}

status = defwInitCbk(fout);
CHECK_STATUS(status);
status = defwVersion (5, 7);
CHECK_STATUS(status);
status = defwDividerChar(":");
CHECK_STATUS(status);
status = defwBusBitChars("[]");
CHECK_STATUS(status);
status = defwDesignName("muk");
CHECK_STATUS(status);
status = defwTechnology("muk");
CHECK_STATUS(status);
status = defwArray("core_array");
```

```
  CHECK_STATUS(status);
  status = defwFloorplan("DEFAULT");
  CHECK_STATUS(status);
  status = defwUnits(100);
  CHECK_STATUS(status);

  // initalize
  status = defwNewLine();
  CHECK_STATUS(status);

  // history
  status = defwHistory("Corrected STEP for ROW_9 and added ROW_10 of SITE CORE1
(def)");
  CHECK_STATUS(status);
  status = defwHistory("Removed NONDEFAULTRULE from the net XX100 (def)");
  CHECK_STATUS(status);
  status = defwHistory("Changed some cell orientations (def)");
  CHECK_STATUS(status);
  status = defwNewLine();
  CHECK_STATUS(status);

  // PROPERTYDEFINITIONS
  status = defwStartPropDef();
  CHECK_STATUS(status);
  defwAddComment("defwPropDef is broken into 3 routines, defwStringPropDef");
  defwAddComment("defwIntPropDef, and defwRealPropDef");
  status = defwStringPropDef("REGION", "scum", 0, 0, 0 );
  CHECK_STATUS(status);
  status = defwIntPropDef("REGION", "center", 0, 0, 0);
  CHECK_STATUS(status);
  status = defwRealPropDef("REGION", "area", 0, 0, 0);
  CHECK_STATUS(status);
  status = defwStringPropDef("GROUP", "ggrp", 0, 0, 0);
  CHECK_STATUS(status);
  status = defwIntPropDef("GROUP", "site", 0, 25, 0);
  CHECK_STATUS(status);
  status = defwRealPropDef("GROUP", "maxarea", 0, 0, 0);
  CHECK_STATUS(status);
  status = defwStringPropDef("COMPONENT", "cc", 0, 0, 0);
  CHECK_STATUS(status);
  status = defwIntPropDef("COMPONENT", "index", 0, 0, 0);
```

```
CHECK_STATUS(status);
status = defwRealPropDef("COMPONENT", "size", 0, 0, 0);
CHECK_STATUS(status);
status = defwIntPropDef("NET", "alt", 0, 0, 0);
CHECK_STATUS(status);
status = defwStringPropDef("NET", "lastName", 0, 0, 0);
CHECK_STATUS(status);
status = defwRealPropDef("NET", "length", 0, 0, 0);
CHECK_STATUS(status);
status = defwStringPropDef("SPECIALNET", "contype", 0, 0, 0);
CHECK_STATUS(status);
status = defwIntPropDef("SPECIALNET", "ind", 0, 0, 0);
CHECK_STATUS(status);
status = defwRealPropDef("SPECIALNET", "maxlength", 0, 0, 0);
CHECK_STATUS(status);
status = defwStringPropDef("DESIGN", "title", 0, 0, "Buffer");
CHECK_STATUS(status);
status = defwIntPropDef("DESIGN", "priority", 0, 0, 14);
CHECK_STATUS(status);
status = defwRealPropDef("DESIGN", "howbig", 0, 0, 15.16);
CHECK_STATUS(status);
status = defwRealPropDef("ROW", "minlength", 1.0, 100.0, 0);
CHECK_STATUS(status);
status = defwStringPropDef("ROW", "firstName", 0, 0, 0);
CHECK_STATUS(status);
status = defwIntPropDef("ROW", "idx", 0, 0, 0);
CHECK_STATUS(status);
status = defwIntPropDef("COMPONENTPIN", "dpIgnoreTerm", 0, 0, 0);
CHECK_STATUS(status);
status = defwStringPropDef("COMPONENTPIN", "dpBit", 0, 0, 0);
CHECK_STATUS(status);
status = defwRealPropDef("COMPONENTPIN", "realProperty", 0, 0, 0);
CHECK_STATUS(status);
status = defwStringPropDef("NET", "IGNOREOPTIMIZATION", 0, 0, 0);
CHECK_STATUS(status);
status = defwStringPropDef("SPECIALNET", "IGNOREOPTIMIZATION", 0, 0, 0);
CHECK_STATUS(status);
status = defwRealPropDef("NET", "FREQUENCY", 0, 0, 0);
CHECK_STATUS(status);
status = defwRealPropDef("SPECIALNET", "FREQUENCY", 0, 0, 0);
CHECK_STATUS(status);
```

```
status = defwStringPropDef("NONDEFAULTRULE", "ndprop1", 0, 0, 0);
CHECK_STATUS(status);
status = defwIntPropDef("NONDEFAULTRULE", "ndprop2", 0, 0, 0);
CHECK_STATUS(status);
status = defwRealPropDef("NONDEFAULTRULE", "ndprop3", 0, 0, 0.009);
CHECK_STATUS(status);
status = defwRealPropDef("NONDEFAULTRULE", "ndprop4", .1, 1.0, 0);
CHECK_STATUS(status);
status = defwEndPropDef();
CHECK_STATUS(status);

// DIEAREA
xPoints = (int*)malloc(sizeof(int)*6);
yPoints = (int*)malloc(sizeof(int)*6);
xPoints[0] = 2000;
yPoints[0] = 2000;
xPoints[1] = 3000;
yPoints[1] = 3000;
xPoints[2] = 4000;
yPoints[2] = 4000;
xPoints[3] = 5000;
yPoints[3] = 5000;
xPoints[4] = 6000;
yPoints[4] = 6000;
xPoints[5] = 7000;
yPoints[5] = 7000;
status = defwDieAreaList(6, xPoints, yPoints);
CHECK_STATUS(status);
free((char*)xPoints);
free((char*)yPoints);

status = defwNewLine();
CHECK_STATUS(status);

// ROW
status = defwRow("ROW_9", "CORE", -177320, -111250, 6, 911, 1, 360, 0);
CHECK_STATUS(status);
status = defwRealProperty("minlength", 50.5);
CHECK_STATUS(status);
status = defwStringProperty("firstName", "Only");
CHECK_STATUS(status);
```

```
status = defwIntProperty("idx", 1);
CHECK_STATUS(status);
status = defwRowStr("ROW_10", "CORE1", -19000, -11000, "FN", 1, 100, 0, 600);
CHECK_STATUS(status);
status = defwRowStr("ROW_11", "CORE1", -19000, -11000, "FN", 1, 100, 0, 0);
CHECK_STATUS(status);
status = defwRow("ROW_12", "CORE1", -19000, -11000, 3, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwRowStr("ROW_13", "CORE1", -19000, -11000, "FN", 0, 0, 0, 0);
CHECK_STATUS(status);

// TRACKS
layers = (const char**)malloc(sizeof(char*)*1);
layers[0] = strdup("M1");
status = defwTracks("X", 3000, 40, 120, 1, layers);
CHECK_STATUS(status);
free((char*)layers[0]);
layers[0] = strdup("M2");
status = defwTracks("Y", 5000, 10, 20, 1,layers);
CHECK_STATUS(status);
free((char*)layers[0]);
free((char*)layers);
status = defwNewLine();
CHECK_STATUS(status);

// GCELLGRID
status = defwGcellGrid("X", 0, 100, 600);
CHECK_STATUS(status);
status = defwGcellGrid("Y", 10, 120, 400);
CHECK_STATUS(status);
status = defwNewLine();
CHECK_STATUS(status);

// CANPLACE
status = defwCanPlaceStr("dp", 45, 64, "N", 35, 1, 39, 1);
CHECK_STATUS(status);

status = defwCanPlace("dp", 45, 64, 1, 35, 1, 39, 1);
CHECK_STATUS(status);

// CANNOTOCCUPY
```

```
status = defwCannotOccupyStr("dp", 54, 44, "S", 55, 2, 45, 3);
CHECK_STATUS(status);

// VIAS
status = defwStartVias(7);
CHECK_STATUS(status);
status = defwViaName("VIA_ARRAY");
CHECK_STATUS(status);
status = defwViaPattern("P1-435-543-IJ1FS");
CHECK_STATUS(status);
status = defwViaRect("M1", -40, -40, 40, 40);
CHECK_STATUS(status);
status = defwViaRect("V1", -40, -40, 40, 40);
CHECK_STATUS(status);
status = defwViaRect("M2", -50, -50, 50, 50);
CHECK_STATUS(status);
status = defwOneViaEnd();
CHECK_STATUS(status);
status = defwViaName("VIA_ARRAY1");
CHECK_STATUS(status);
status = defwViaRect("M1", -40, -40, 40, 40);
CHECK_STATUS(status);
status = defwViaRect("V1", -40, -40, 40, 40);
CHECK_STATUS(status);
status = defwViaRect("M2", -50, -50, 50, 50);
CHECK_STATUS(status);
status = defwOneViaEnd();
CHECK_STATUS(status);
status = defwViaName("myUnshiftedVia");
CHECK_STATUS(status);
status = defwViaViarule("myViaRule", 20, 20, "metal1", "cut12", "metal2",
                        5, 5, 0, 4, 0, 1);
CHECK_STATUS(status);
status = defwViaViaruleRowCol(2, 3);
CHECK_STATUS(status);
status = defwOneViaEnd();
CHECK_STATUS(status);
status = defwViaName("via2");
CHECK_STATUS(status);
status = defwViaViarule("viaRule2", 5, 6, "botLayer2", "cutLayer2",
                        "topLayer2", 6, 6, 1, 4, 1, 4);
```

```
CHECK_STATUS(status);
status = defwViaViaruleOrigin(10, -10);
CHECK_STATUS(status);
status = defwViaViaruleOffset(0, 0, 20, -20);
CHECK_STATUS(status);
status = defwViaViarulePattern("2_F0_2_F8_1_78");
CHECK_STATUS(status);
status = defwOneViaEnd();
CHECK_STATUS(status);

status = defwViaName("via3");
CHECK_STATUS(status);
status = defwViaPattern("P2-435-543-IJ1FS");
CHECK_STATUS(status);
status = defwViaRect("M2", -40, -40, 40, 40);
CHECK_STATUS(status);
status = defwOneViaEnd();
CHECK_STATUS(status);

xP = (double*)malloc(sizeof(double)*6);
yP = (double*)malloc(sizeof(double)*6);
xP[0] = -2.1;
yP[0] = -1.0;
xP[1] = -2;
yP[1] = 1;
xP[2] = 2.1;
yP[2] = 1.0;
xP[3] = 2.0;
yP[3] = -1.0;
status = defwViaName("via4");
CHECK_STATUS(status);
status = defwViaPolygon("M3", 4, xP, yP);
CHECK_STATUS(status);
status = defwViaRect("M4", -40, -40, 40, 40);
CHECK_STATUS(status);
xP[0] = 100;
yP[0] = 100;
xP[1] = 200;
yP[1] = 200;
xP[2] = 300;
yP[2] = 300;
```

```
xP[3] = 400;
yP[3] = 400;
xP[4] = 500;
yP[4] = 500;
xP[5] = 600;
yP[5] = 600;
status = defwViaPolygon("M5", 6, xP, yP);
CHECK_STATUS(status);
status = defwOneViaEnd();
CHECK_STATUS(status);

xP[0] = 200;
yP[0] = 200;
xP[1] = 300;
yP[1] = 300;
xP[2] = 400;
yP[2] = 500;
xP[3] = 100;
yP[3] = 300;
xP[4] = 300;
yP[4] = 200;
status = defwViaName("via5");
CHECK_STATUS(status);
status = defwViaPolygon("M6", 5, xP, yP);
CHECK_STATUS(status);
status = defwOneViaEnd();
CHECK_STATUS(status);
free((char*)xP);
free((char*)yP);
status = defwEndVias();
CHECK_STATUS(status);

// REGIONS
status = defwStartRegions(2);
CHECK_STATUS(status);
status = defwRegionName("region1");
CHECK_STATUS(status);
status = defwRegionPoints(-500, -500, 300, 100);
CHECK_STATUS(status);
status = defwRegionPoints(500, 500, 1000, 1000);
CHECK_STATUS(status);
```

```
status = defwRegionType("FENCE");
CHECK_STATUS(status);
status = defwStringProperty("scum", "on top");
CHECK_STATUS(status);
status = defwIntProperty("center", 250);
CHECK_STATUS(status);
status = defwIntProperty("area", 730000);
CHECK_STATUS(status);
status = defwRegionName("region2");
CHECK_STATUS(status);
status = defwRegionPoints(4000, 0, 5000, 1000);
CHECK_STATUS(status);
status = defwStringProperty("scum", "on bottom");
CHECK_STATUS(status);
status = defwEndRegions();
CHECK_STATUS(status);

// COMPONENTS
foreigns = (const char**)malloc(sizeof(char*)*2);
foreignX = (int*)malloc(sizeof(int)*2);
foreignY = (int*)malloc(sizeof(int)*2);
foreignOrient = (int*)malloc(sizeof(int)*2);
foreignOrientStr = (const char**)malloc(sizeof(char*)*2);
status = defwStartComponents(11);
CHECK_STATUS(status);
status = defwComponent("Z38A01", "DFF3", 0, NULL, NULL, NULL, NULL, NULL,
                       0, NULL, NULL, NULL, NULL, "PLACED", 18592, 5400, 6, 0,
                       NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwComponentHalo(100, 0, 50, 200);
CHECK_STATUS(status);
status = defwComponentStr("Z38A03", "DFF3", 0, NULL, NULL, NULL, NULL, NULL,
                       0, NULL, NULL, NULL, NULL, "PLACED", 16576, 45600,
                       "FS", 0, NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwComponentHalo(200, 2, 60, 300);
CHECK_STATUS(status);
status = defwComponent("Z38A05", "DFF3", 0, NULL, NULL, NULL, NULL, NULL,
                       0, NULL, NULL, NULL, NULL, "PLACED", 51520, 9600, 6, 0,
                       NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
```

```
status = defwComponent("|i0", "INV_B", 0, NULL, "INV", NULL, NULL, NULL,
                       0, NULL, NULL, NULL, NULL, NULL, 0, 0, -1, 0,
                       "region1", 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwComponentHaloSoft(100, 0, 50, 200);
CHECK_STATUS(status);
status = defwComponent("|i1", "INV_B", 0, NULL, "INV", NULL, NULL, NULL,
                       0, NULL, NULL, NULL, NULL, "UNPLACED", 1000, 1000, 0,
                       0, NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwComponent("cell1", "CHM6A", 0, NULL, NULL, "generator", NULL,
                       "USER", 0, NULL, NULL, NULL, NULL, "FIXED", 0, 10, 0,
                       100.4534535, NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwComponent("cell2", "CHM6A", 0, NULL, NULL, NULL, NULL,
                       "NETLIST", 0, NULL, NULL, NULL, NULL, "COVER", 120,
                       10, 4, 2, NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
foreigns[0] = strdup("gds2name");
foreignX[0] = -500;
foreignY[0] = -500;
foreignOrient[0] = 3;
status = defwComponent("cell3", "CHM6A", 0, NULL, NULL, NULL, NULL,
                       "TIMING", 1, foreigns, foreignX, foreignY,
                       foreignOrient, "PLACED", 240,
                       10, 0, 0, "region1", 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwComponentRouteHalo(100, "metal1", "metal3");
CHECK_STATUS(status);
free((char*)foreigns[0]);
foreigns[0] = strdup("gds3name");
foreignX[0] = -500;
foreignY[0] = -500;
foreignOrientStr[0] = strdup("FW");
foreigns[1] = strdup("gds4name");
foreignX[1] = -300;
foreignY[1] = -300;
foreignOrientStr[1] = strdup("FS");
status = defwComponentStr("cell4", "CHM3A", 0, NULL, "CHM6A", NULL, NULL,
                       "DIST", 2, foreigns, foreignX, foreignY,
                       foreignOrientStr, "PLACED", 360,
```

```
                        10, "W", 0, "region2", 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwComponentHaloSoft(100, 0, 50, 200);
CHECK_STATUS(status);
status = defwStringProperty("cc", "This is the copy list");
CHECK_STATUS(status);
status = defwIntProperty("index", 9);
CHECK_STATUS(status);
status = defwRealProperty("size", 7.8);
CHECK_STATUS(status);
status = defwComponent("scancell1", "CHK3A", 0, NULL, NULL, NULL, NULL,
                        NULL, 0, NULL, NULL, NULL, NULL, "PLACED", 500,
                        10, 7, 0, NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwComponent("scancell2", "CHK3A", 0, NULL, NULL, NULL, NULL,
                        NULL, 0, NULL, NULL, NULL, NULL, "PLACED", 700,
                        10, 6, 0, NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwEndComponents();
CHECK_STATUS(status);
free((char*)foreigns[0]);
free((char*)foreigns[1]);
free((char*)foreigns);
free((char*)foreignX);
free((char*)foreignY);
free((char*)foreignOrient);
free((char*)foreignOrientStr[0]);
free((char*)foreignOrientStr[1]);
free((char*)foreignOrientStr);

xP = (double*)malloc(sizeof(double)*6);
yP = (double*)malloc(sizeof(double)*6);
xP[0] = 2.1;
yP[0] = 2.1;
xP[1] = 3.1;
yP[1] = 3.1;
xP[2] = 4.1;
yP[2] = 4.1;
xP[3] = 5.1;
yP[3] = 5.1;
xP[4] = 6.1;
```

```
yP[4] = 6.1;
xP[5] = 7.1;
yP[5] = 7.1;

// PINS
status = defwStartPins(11);
CHECK_STATUS(status);
status = defwPin("scanpin", "net1", 0, "INPUT", NULL, NULL, 0, 0, -1, NULL,
                 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwPinPolygon("metal1", 0, 1000, 6, xP, yP);
CHECK_STATUS(status);
status = defwPinNetExpr("power1 VDD1");
CHECK_STATUS(status);
status = defwPin("pin0", "net1", 0, "INPUT", "SCAN", NULL, 0, 0, -1, NULL,
                 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwPinStr("pin0.5", "net1", 0, "INPUT", "RESET", "FIXED", 0, 0, "S",
                    NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwPinPolygon("metal2", 0, 0, 4, xP, yP);
CHECK_STATUS(status);
status = defwPinLayer("metal3", 500, 0, -5000, -100, -4950, -90);
CHECK_STATUS(status);
status = defwPin("pin1", "net1", 1, NULL, "POWER", NULL, 0, 0, -1, "M1",
                 -5000, -100, -4950, -90);
CHECK_STATUS(status);
status = defwPinAntennaPinPartialMetalArea(4580, "M1");
CHECK_STATUS(status);
status = defwPinAntennaPinPartialMetalArea(4580, "M11");
CHECK_STATUS(status);
status = defwPinAntennaPinPartialMetalArea(4580, "M12");
CHECK_STATUS(status);
status = defwPinAntennaPinGateArea(4580, "M2");
CHECK_STATUS(status);
status = defwPinAntennaPinDiffArea(4580, "M3");
CHECK_STATUS(status);
status = defwPinAntennaPinDiffArea(4580, "M31");
CHECK_STATUS(status);
status = defwPinAntennaPinMaxAreaCar(5000, "L1");
CHECK_STATUS(status);
```

```
status = defwPinAntennaPinMaxSideAreaCar(5000, "M4");
CHECK_STATUS(status);
status = defwPinAntennaPinPartialCutArea(4580, "M4");
CHECK_STATUS(status);
status = defwPinAntennaPinMaxCutCar(5000, "L1");
CHECK_STATUS(status);
status = defwPin("pin2", "net2", 0, "INPUT", "SIGNAL", NULL, 0, 0, -1, "M1",
                 -5000, 0, -4950, 10);
CHECK_STATUS(status);
status = defwPinLayer("M1", 500, 0, -5000, 0, -4950, 10);
CHECK_STATUS(status);
status = defwPinPolygon("M2", 0, 0, 4, xP, yP);
CHECK_STATUS(status);
status = defwPinPolygon("M3", 0, 0, 3, xP, yP);
CHECK_STATUS(status);
status = defwPinLayer("M4", 0, 500, 0, 100, -400, 100);
CHECK_STATUS(status);
status = defwPinSupplySensitivity("vddpin1");
CHECK_STATUS(status);
status = defwPinGroundSensitivity("gndpin1");
CHECK_STATUS(status);
status = defwPinAntennaPinPartialMetalArea(5000, NULL);
CHECK_STATUS(status);
status = defwPinAntennaPinPartialMetalSideArea(4580, "M2");
CHECK_STATUS(status);
status = defwPinAntennaPinGateArea(5000, NULL);
CHECK_STATUS(status);
status = defwPinAntennaPinPartialCutArea(5000, NULL);
CHECK_STATUS(status);
status = defwPin("INBUS[1]", "|INBUS[1]", 0, "INPUT", "SIGNAL", "FIXED",
                 45, -2160, 0, "M2", 0, 0, 30, 135);
CHECK_STATUS(status);
status = defwPinLayer("M2", 0, 0, 0, 0, 30, 135);
CHECK_STATUS(status);
status = defwPinAntennaPinPartialMetalArea(1, "M1");
CHECK_STATUS(status);
status = defwPinAntennaPinPartialMetalSideArea(2, "M1");
CHECK_STATUS(status);
status = defwPinAntennaPinDiffArea(4, "M2");
CHECK_STATUS(status);
status = defwPinAntennaPinPartialCutArea(5, "V1");
```

```
CHECK_STATUS(status);
status = defwPinAntennaModel("OXIDE1");
CHECK_STATUS(status);
status = defwPinAntennaPinGateArea(3, "M1");
CHECK_STATUS(status);
status = defwPinAntennaPinMaxAreaCar(6, "M2");
CHECK_STATUS(status);
status = defwPinAntennaPinMaxSideAreaCar(7, "M2");
CHECK_STATUS(status);
status = defwPinAntennaPinMaxCutCar(8, "V1");
CHECK_STATUS(status);
status = defwPinAntennaModel("OXIDE2");
CHECK_STATUS(status);
status = defwPinAntennaPinGateArea(30, "M1");
CHECK_STATUS(status);
status = defwPinAntennaPinMaxAreaCar(60, "M2");
CHECK_STATUS(status);
status = defwPinAntennaPinMaxSideAreaCar(70, "M2");
CHECK_STATUS(status);
status = defwPinAntennaPinMaxCutCar(80, "V1");
CHECK_STATUS(status);
status = defwPin("INBUS<0>", "|INBUS<0>", 0, "INPUT", "SIGNAL", "PLACED",
                 -45, 2160, 1, "M2", 0, 0, 30, 134);
CHECK_STATUS(status);
status = defwPinLayer("M2", 0, 1000, 0, 0, 30, 134);
CHECK_STATUS(status);
status = defwPin("OUTBUS<1>", "|OUTBUS<1>", 0, "OUTPUT", "SIGNAL", "COVER",
                 2160, 645, 2, "M1", 0, 0, 30, 135);
CHECK_STATUS(status);
status = defwPinLayer("M1", 0, 0, 0, 0, 30, 134);
CHECK_STATUS(status);
status = defwPinNetExpr("gnd1 GND");
CHECK_STATUS(status);
status = defwPin("VDD", "VDD", 1, "INOUT", "POWER", NULL, 0, 0, -1, NULL,
                 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwPin("BUSA[0]", "BUSA[0]", 0, "INPUT", "SIGNAL", "PLACED",
                 0, 2500, 1, NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwPinLayer("M1", 0, 0, -25, 0, 25, 50);
CHECK_STATUS(status);
```

```
status = defwPinLayer("M2", 0, 0, -10, 0, 10, 75);
CHECK_STATUS(status);
status = defwPinVia("via12", 0, 25);
CHECK_STATUS(status);
status = defwPin("VDD", "VDD", 1, "INOUT", "POWER", NULL,
                 0, 0, -1, NULL, 0, 0, 0, 0);
CHECK_STATUS(status);
status = defwPinPort();
CHECK_STATUS(status);
status = defwPinPortLayer("M2", 0, 0, -25, 0, 25, 50);
CHECK_STATUS(status);
status = defwPinPortLocation("PLACED", 0, 2500, "S");
CHECK_STATUS(status);
status = defwPinPort();
CHECK_STATUS(status);
status = defwPinPortLayer("M1", 0, 0, -25, 0, 25, 50);
CHECK_STATUS(status);
status = defwPinPortLocation("COVER", 0, 2500, "S");
CHECK_STATUS(status);
status = defwPinPort();
CHECK_STATUS(status);
status = defwPinPortLayer("M1", 0, 0, -25, 0, 25, 50);
CHECK_STATUS(status);
status = defwPinPortLocation("FIXED", 0, 2500, "S");
CHECK_STATUS(status);

status = defwEndPins();
CHECK_STATUS(status);

free((char*)xP);
free((char*)yP);

// PINPROPERTIES
status = defwStartPinProperties(2);
CHECK_STATUS(status);
status = defwPinProperty("cell1", "PB1");
CHECK_STATUS(status);
status = defwStringProperty("dpBit", "1");
CHECK_STATUS(status);
status = defwRealProperty("realProperty", 3.4);
CHECK_STATUS(status);
```

```
status = defwPinProperty("cell2", "vdd");
CHECK_STATUS(status);
status = defwIntProperty("dpIgnoreTerm", 2);
CHECK_STATUS(status);
status = defwEndPinProperties();
CHECK_STATUS(status);

// SPECIALNETS
status = defwStartSpecialNets(7);
CHECK_STATUS(status);
status = defwSpecialNet("net1");
CHECK_STATUS(status);
status = defwSpecialNetConnection("cell1", "VDD", 0);
CHECK_STATUS(status);
status = defwSpecialNetConnection("cell2", "VDD", 0);
CHECK_STATUS(status);
status = defwSpecialNetConnection("cell3", "VDD", 0);
CHECK_STATUS(status);
status = defwSpecialNetConnection("cell4", "VDD", 0);
CHECK_STATUS(status);
status = defwSpecialNetWidth("M1", 200);
CHECK_STATUS(status);
status = defwSpecialNetWidth("M2", 300);
CHECK_STATUS(status);
status = defwSpecialNetVoltage(3.2);
CHECK_STATUS(status);
status = defwSpecialNetSpacing("M1", 200, 190, 210);
CHECK_STATUS(status);
status = defwSpecialNetSource("TIMING");
CHECK_STATUS(status);
status = defwSpecialNetOriginal("VDD");
CHECK_STATUS(status);
status = defwSpecialNetUse("POWER");
CHECK_STATUS(status);
status = defwSpecialNetWeight(30);
CHECK_STATUS(status);
status = defwStringProperty("contype", "star");
CHECK_STATUS(status);
status = defwIntProperty("ind", 1);
CHECK_STATUS(status);
status = defwRealProperty("maxlength", 12.13);
```

```
CHECK_STATUS(status);
status = defwSpecialNetEndOneNet();
CHECK_STATUS(status);
status = defwSpecialNet("VSS");
CHECK_STATUS(status);
status = defwSpecialNetConnection("cell1", "GND", 1);
CHECK_STATUS(status);
status = defwSpecialNetConnection("cell2", "GND", 0);
CHECK_STATUS(status);
status = defwSpecialNetConnection("cell3", "GND", 1);
CHECK_STATUS(status);
status = defwSpecialNetConnection("cell4", "GND", 0);
CHECK_STATUS(status);
status = defwSpecialNetUse("SCAN");
CHECK_STATUS(status);
status = defwSpecialNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwSpecialNetPathLayer("M1");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(250);
CHECK_STATUS(status);
status = defwSpecialNetPathShape("IOWIRE");
CHECK_STATUS(status);
coorX = (const char**)malloc(sizeof(char*)*3);
coorY = (const char**)malloc(sizeof(char*)*3);
coorValue = (const char**)malloc(sizeof(char*)*3);
coorX[0] = strdup("5");
coorY[0] = strdup("15");
coorValue[0] = NULL;
coorX[1] = strdup("125");
coorY[1] = strdup("*");
coorValue[1] = strdup("235");
coorX[2] = strdup("245");
coorY[2] = strdup("*");
coorValue[2] = strdup("255");
status = defwSpecialNetPathPointWithWireExt(3, coorX, coorY, coorValue);
CHECK_STATUS(status);
status = defwSpecialNetPathEnd();
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
```

```
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorValue[0]);
free((char*)coorValue[1]);
free((char*)coorValue[2]);
free((char*)coorValue);
status = defwSpecialNetShieldStart("my_net");
CHECK_STATUS(status);
status = defwSpecialNetShieldLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetShieldWidth(90);
CHECK_STATUS(status);
status = defwSpecialNetShieldShape("STRIPE");
CHECK_STATUS(status);
coorX[0] = strdup("14100");
coorY[0] = strdup("342440");
coorX[1] = strdup("13920");
coorY[1] = strdup("*");
status = defwSpecialNetShieldPoint(2, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetShieldVia("M2_TURN");
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
coorX[0] = strdup("*");
coorY[0] = strdup("263200");
status = defwSpecialNetShieldPoint(1, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetShieldVia("M1_M2");
CHECK_STATUS(status);
status = defwSpecialNetShieldViaData(10, 20, 1000, 2000);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
coorX[0] = strdup("2400");
coorY[0] = strdup("*");
status = defwSpecialNetShieldPoint(1, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetShieldEnd();
CHECK_STATUS(status);
status = defwSpecialNetShieldStart("my_net1");
```

```
CHECK_STATUS(status);
status = defwSpecialNetShieldLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetShieldWidth(90);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
coorX[0] = strdup("14100");
coorY[0] = strdup("342440");
coorX[1] = strdup("13920");
coorY[1] = strdup("*");
status = defwSpecialNetShieldPoint(2, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetShieldVia("M2_TURN");
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
coorX[0] = strdup("*");
coorY[0] = strdup("263200");
status = defwSpecialNetShieldPoint(1, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetShieldVia("M1_M2");
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
coorX[0] = strdup("2400");
coorY[0] = strdup("*");
status = defwSpecialNetShieldPoint(1, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetShieldEnd();
CHECK_STATUS(status);
status = defwSpecialNetPattern("STEINER");
CHECK_STATUS(status);
status = defwSpecialNetEstCap(100);
CHECK_STATUS(status);
status = defwSpecialNetEndOneNet();
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
```

```
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorX[2]);
free((char*)coorY[2]);
status = defwSpecialNet("VDD");
CHECK_STATUS(status);
status = defwSpecialNetConnection("*", "VDD", 0);
CHECK_STATUS(status);
status = defwSpecialNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwSpecialNetPathLayer("metal2");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(100);
CHECK_STATUS(status);
status = defwSpecialNetPathShape("RING");
CHECK_STATUS(status);
status = defwSpecialNetPathStyle(1);
CHECK_STATUS(status);
coorX[0] = strdup("0");
coorY[0] = strdup("0");
coorX[1] = strdup("100");
coorY[1] = strdup("100");
coorX[2] = strdup("200");
coorY[2] = strdup("100");
status = defwSpecialNetPathPoint(3, coorX, coorY);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorX[2]);
free((char*)coorY[2]);
status = defwSpecialNetPathStart("NEW");
CHECK_STATUS(status);
status = defwSpecialNetPathLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(270);
CHECK_STATUS(status);
status = defwSpecialNetPathShape("PADRING");
CHECK_STATUS(status);
coorX[0] = strdup("-45");
```

```
coorY[0] = strdup("1350");
coorX[1] = strdup("44865");
coorY[1] = strdup("*");
status = defwSpecialNetPathPoint(2, coorX, coorY);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwSpecialNetPathStart("NEW");
CHECK_STATUS(status);
status = defwSpecialNetPathLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(270);
CHECK_STATUS(status);
coorX[0] = strdup("-45");
coorY[0] = strdup("1350");
coorX[1] = strdup("44865");
coorY[1] = strdup("*");
status = defwSpecialNetPathPoint(2, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetPathEnd();
CHECK_STATUS(status);
status = defwSpecialNetEndOneNet();
CHECK_STATUS(status);
status = defwSpecialNet("CLOCK");
CHECK_STATUS(status);
status = defwSpecialNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwSpecialNetPathLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(200);
CHECK_STATUS(status);
status = defwSpecialNetPathShape("BLOCKRING");
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
coorX[0] = strdup("-45");
coorY[0] = strdup("1350");
```

```
coorX[1] = strdup("44865");
coorY[1] = strdup("*");
status = defwSpecialNetPathPoint(2, coorX, coorY);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwSpecialNetPathStart("NEW");
CHECK_STATUS(status);
status = defwSpecialNetPathLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(270);
CHECK_STATUS(status);
coorX[0] = strdup("-45");
coorY[0] = strdup("1350");
coorX[1] = strdup("44865");
coorY[1] = strdup("*");
status = defwSpecialNetPathPoint(2, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetPathEnd();
CHECK_STATUS(status);
status = defwSpecialNetEndOneNet();
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwSpecialNet("VCC");
CHECK_STATUS(status);
status = defwSpecialNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwSpecialNetPathLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(200);
CHECK_STATUS(status);
status = defwSpecialNetPathShape("DRCFILL");
CHECK_STATUS(status);
coorX[0] = strdup("-45");
coorY[0] = strdup("1350");
coorX[1] = strdup("44865");
```

```
coorY[1] = strdup("*");
status = defwSpecialNetPathPoint(2, coorX, coorY);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwSpecialNetPathStart("NEW");
CHECK_STATUS(status);
status = defwSpecialNetPathLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(270);
CHECK_STATUS(status);
status = defwSpecialNetPathShape("STRIPE");
CHECK_STATUS(status);
coorX[0] = strdup("-45");
coorY[0] = strdup("1350");
coorX[1] = strdup("44865");
coorY[1] = strdup("*");
status = defwSpecialNetPathPoint(2, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetPathEnd();
CHECK_STATUS(status);
status = defwSpecialNetEndOneNet();
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwSpecialNet("n1");
CHECK_STATUS(status);
status = defwSpecialNetConnection("PIN", "n1", 0);
CHECK_STATUS(status);
status = defwSpecialNetConnection("driver1", "in", 0);
CHECK_STATUS(status);
status = defwSpecialNetConnection("bumpa1", "bumppin", 0);
CHECK_STATUS(status);
status = defwSpecialNetFixedbump();
CHECK_STATUS(status);
status = defwSpecialNetPathStart("ROUTED");
CHECK_STATUS(status);
```

```
status = defwSpecialNetPathLayer("M2");
CHECK_STATUS(status);
status = defwSpecialNetPathWidth(200);
CHECK_STATUS(status);
status = defwSpecialNetPathShape("FILLWIREOPC");
CHECK_STATUS(status);
coorX[0] = strdup("-45");
coorY[0] = strdup("1350");
coorX[1] = strdup("44865");
coorY[1] = strdup("*");
status = defwSpecialNetPathPoint(2, coorX, coorY);
CHECK_STATUS(status);
status = defwSpecialNetPathEnd();
CHECK_STATUS(status);
status = defwSpecialNetEndOneNet();
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorX);
free((char*)coorY);

status = defwSpecialNet("VSS1");
CHECK_STATUS(status);
status = defwSpecialNetUse("POWER");
CHECK_STATUS(status);
xP = (double*)malloc(sizeof(double)*6);
yP = (double*)malloc(sizeof(double)*6);
xP[0] = 2.1;
yP[0] = 2.1;
xP[1] = 3.1;
yP[1] = 3.1;
xP[2] = 4.1;
yP[2] = 4.1;
xP[3] = 5.1;
yP[3] = 5.1;
xP[4] = 6.1;
yP[4] = 6.1;
xP[5] = 7.1;
yP[5] = 7.1;
```

```
status = defwSpecialNetPolygon("metal1", 4, xP, yP);
CHECK_STATUS(status);
status = defwSpecialNetPolygon("metal1", 6, xP, yP);
CHECK_STATUS(status);
status = defwSpecialNetRect("metal1", 0, 0, 100, 200);
CHECK_STATUS(status);
status = defwSpecialNetRect("metal2", 1, 1, 100, 200);
CHECK_STATUS(status);
status = defwSpecialNetEndOneNet();
CHECK_STATUS(status);
free((char*)xP);
free((char*)yP);
status = defwEndSpecialNets();
CHECK_STATUS(status);

// NETS
status = defwStartNets(12);
CHECK_STATUS(status);
status = defwNet("net1");
CHECK_STATUS(status);
status = defwNetConnection("Z38A01", "Q", 0);
CHECK_STATUS(status);
status = defwNetConnection("Z38A03", "Q", 0);
CHECK_STATUS(status);
status = defwNetConnection("Z38A05", "Q", 0);
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwNet("net2");
CHECK_STATUS(status);
status = defwNetConnection("cell1", "PB1", 0);
CHECK_STATUS(status);
status = defwNetConnection("cell2", "PB1", 0);
CHECK_STATUS(status);
status = defwNetConnection("cell3", "PB1", 0);
CHECK_STATUS(status);
status = defwNetEstCap(200);
CHECK_STATUS(status);
status = defwNetWeight(2);
CHECK_STATUS(status);
```

```
status = defwNetVpin("P1", NULL, 0, 0, 0, 0, "PLACED", 54, 64, 3);
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwNet("net3");
CHECK_STATUS(status);
status = defwNetConnection("cell4", "PA3", 0);
CHECK_STATUS(status);
status = defwNetConnection("cell2", "P10", 0);
CHECK_STATUS(status);
status = defwNetXtalk(30);
CHECK_STATUS(status);
status = defwNetOriginal("extra_crispy");
CHECK_STATUS(status);
status = defwNetSource("USER");
CHECK_STATUS(status);
status = defwNetUse("SIGNAL");
CHECK_STATUS(status);
status = defwNetFrequency(100);
CHECK_STATUS(status);
status = defwIntProperty("alt", 37);
CHECK_STATUS(status);
status = defwStringProperty("lastName", "Unknown");
CHECK_STATUS(status);
status = defwRealProperty("length", 10.11);
CHECK_STATUS(status);
status = defwNetPattern("BALANCED");
CHECK_STATUS(status);
status = defwNetVpinStr("P2", "L1", 45, 54, 3, 46, "FIXED", 23, 12, "FN");
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

coorX = (const char**)malloc(sizeof(char*)*5);
coorY = (const char**)malloc(sizeof(char*)*5);
coorValue = (const char**)malloc(sizeof(char*)*5);
status = defwNet("my_net");
CHECK_STATUS(status);
status = defwNetConnection("I1", "A", 0);
CHECK_STATUS(status);
```

```
status = defwNetConnection("BUF", "Z", 0);
CHECK_STATUS(status);
status = defwNetNondefaultRule("RULE1");
CHECK_STATUS(status);
status = defwNetUse("RESET");
CHECK_STATUS(status);
status = defwNetShieldnet("VSS");
CHECK_STATUS(status);
status = defwNetShieldnet("VDD");
CHECK_STATUS(status);
status = defwNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwNetPathLayer("M2", 0, NULL);
CHECK_STATUS(status);
status = defwNetPathStyle(2);
CHECK_STATUS(status);
coorX[0] = strdup("14000");
coorY[0] = strdup("341440");
coorValue[0] = NULL;
coorX[1] = strdup("9600");
coorY[1] = strdup("*");
coorValue[1] = NULL;
coorX[2] = strdup("*");
coorY[2] = strdup("282400");
coorValue[2] = NULL;
status = defwNetPathPoint(3, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("nd1VIA12");
CHECK_STATUS(status);
coorX[0] = strdup("2400");
coorY[0] = strdup("*");
coorValue[0] = NULL;
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwNetPathStart("NEW");
```

```
CHECK_STATUS(status);
status = defwNetPathLayer("M1", 1, NULL);
CHECK_STATUS(status);
status = defwNetPathStyle(4);
CHECK_STATUS(status);
coorX[0] = strdup("2400");
coorY[0] = strdup("282400");
coorValue[0] = NULL;
coorX[1] = strdup("240");
coorY[1] = strdup("*");
coorValue[1] = NULL;
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorX[2]);
free((char*)coorY[2]);
status = defwNetPathEnd();
CHECK_STATUS(status);
status = defwNetNoshieldStart("M2");
CHECK_STATUS(status);
coorX[0] = strdup("14100");
coorY[0] = strdup("341440");
coorX[1] = strdup("14000");
coorY[1] = strdup("*");
status = defwNetNoshieldPoint(2, coorX, coorY);
CHECK_STATUS(status);
status = defwNetNoshieldEnd();
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwNet("|INBUS[1]");
CHECK_STATUS(status);
status = defwNetConnection("|i1", "A", 0);
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);
```

```
status = defwNet("|INBUS<0>");
CHECK_STATUS(status);
status = defwNetConnection("|i0", "A", 0);
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwNet("|OUTBUS<1>");
CHECK_STATUS(status);
status = defwNetConnection("|i0", "Z", 0);
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwNet("MUSTJOIN");
CHECK_STATUS(status);
status = defwNetConnection("cell4", "PA1", 0);
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwNet("XX100");
CHECK_STATUS(status);
status = defwNetConnection("Z38A05", "G", 0);
CHECK_STATUS(status);
status = defwNetConnection("Z38A03", "G", 0);
CHECK_STATUS(status);
status = defwNetConnection("Z38A01", "G", 0);
CHECK_STATUS(status);
status = defwNetVpin("V_SUB3_XX100", NULL, -333, -333, 333, 333, "PLACED",
                     189560, 27300, 0);
CHECK_STATUS(status);
status = defwNetVpin("V_SUB2_XX100", NULL, -333, -333, 333, 333, "PLACED",
                     169400, 64500, 0);
CHECK_STATUS(status);
status = defwNetVpin("V_SUB1_XX100", NULL, -333, -333, 333, 333, "PLACED",
                     55160, 31500, 0);
CHECK_STATUS(status);
status = defwNetSubnetStart("SUB1_XX100");
CHECK_STATUS(status);
status = defwNetSubnetPin("Z38A05", "G");
```

```
CHECK_STATUS(status);
status = defwNetSubnetPin("VPIN", "V_SUB1_XX100");
CHECK_STATUS(status);
status = defwNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwNetPathLayer("M1", 0, "RULE1");
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
coorX[0] = strdup("54040");
coorY[0] = strdup("30300");
coorValue[0] = strdup("0");
coorX[1] = strdup("*");
coorY[1] = strdup("30900");
coorValue[1] = NULL;
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwNetPathVia("nd1VIA12");
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("*");
coorValue[0] = strdup("0");
coorX[1] = strdup("56280");
coorY[1] = strdup("*");
coorValue[1] = NULL;
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwNetPathViaWithOrient("nd1VIA23", 6);
CHECK_STATUS(status);
```

```
coorX[0] = strdup("*");
coorY[0] = strdup("31500");
coorValue[0] = NULL;
coorX[1] = strdup("55160");
coorY[1] = strdup("*");
coorValue[1] = NULL;
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwNetPathEnd();
CHECK_STATUS(status);
status = defwNetSubnetEnd();
CHECK_STATUS(status);
status = defwNetSubnetStart("SUB2_XX100");
CHECK_STATUS(status);
status = defwNetSubnetPin("Z38A03", "G");
CHECK_STATUS(status);
status = defwNetSubnetPin("VPIN", "V_SUB2_XX100");
CHECK_STATUS(status);
status = defwNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwNetPathLayer("M1", 0, NULL);
CHECK_STATUS(status);
coorX[0] = strdup("168280");
coorY[0] = strdup("63300");
coorValue[0] = strdup("7");
coorX[1] = strdup("*");
coorY[1] = strdup("64500");
coorValue[1] = NULL;
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwNetPathVia("M1_M2");
CHECK_STATUS(status);
```

```
coorX[0] = strdup("169400");
coorY[0] = strdup("*");
coorValue[0] = strdup("8");
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
status = defwNetPathViaWithOrientStr("M2_M3", "SE");
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
status = defwNetPathEnd();
CHECK_STATUS(status);
status = defwNetSubnetEnd();
CHECK_STATUS(status);
status = defwNetSubnetStart("SUB3_XX100");
CHECK_STATUS(status);
status = defwNetSubnetPin("Z38A01", "G");
CHECK_STATUS(status);
status = defwNetSubnetPin("VPIN", "V_SUB3_XX100");
CHECK_STATUS(status);
status = defwNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwNetPathLayer("M1", 0, NULL);
CHECK_STATUS(status);
coorX[0] = strdup("188400");
coorY[0] = strdup("26100");
coorValue[0] = strdup("0");
coorX[1] = strdup("*");
coorY[1] = strdup("27300");
coorValue[1] = strdup("0");
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorValue[1]);
status = defwNetPathVia("M1_M2");
CHECK_STATUS(status);
coorX[0] = strdup("189560");
```

```
coorY[0] = strdup("*");
coorValue[0] = strdup("0");
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
status = defwNetPathVia("M1_M2");
CHECK_STATUS(status);
status = defwNetPathEnd();
CHECK_STATUS(status);
status = defwNetSubnetEnd();
CHECK_STATUS(status);
status = defwNetSubnetStart("SUB0_XX100");
CHECK_STATUS(status);
status = defwNetSubnetPin("VPIN", "V_SUB1_XX100");
CHECK_STATUS(status);
status = defwNetSubnetPin("VPIN", "V_SUB2_XX100");
CHECK_STATUS(status);
status = defwNetSubnetPin("VPIN", "V_SUB3_XX100");
CHECK_STATUS(status);
status = defwNetNondefaultRule("RULE1");
CHECK_STATUS(status);
status = defwNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwNetPathLayer("M3", 0, NULL);
CHECK_STATUS(status);
coorX[0] = strdup("269400");
coorY[0] = strdup("64500");
coorValue[0] = strdup("0");
coorX[1] = strdup("*");
coorY[1] = strdup("54900");
coorValue[1] = NULL;
coorX[2] = strdup("170520");
coorY[2] = strdup("*");
coorValue[2] = NULL;
coorX[3] = strdup("*");
coorY[3] = strdup("37500");
coorValue[3] = NULL;
coorX[4] = strdup("*");
coorY[4] = strdup("30300");
```

```
coorValue[4] = NULL;
status = defwNetPathPoint(5, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorX[2]);
free((char*)coorY[2]);
free((char*)coorX[3]);
free((char*)coorY[3]);
free((char*)coorX[4]);
free((char*)coorY[4]);
status = defwNetPathVia("nd1VIA23");
CHECK_STATUS(status);
coorX[0] = strdup("171080");
coorY[0] = strdup("*");
coorValue[0] = NULL;
coorX[1] = strdup("17440");
coorY[1] = strdup("0");
coorValue[1] = strdup("0");
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorValue[1]);
status = defwNetPathVia("nd1VIA23");
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("*");
coorValue[0] = NULL;
coorX[1] = strdup("*");
coorY[1] = strdup("26700");
coorValue[1] = strdup("8");
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
```

```
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorValue[1]);
status = defwNetPathVia("nd1VIA23");
CHECK_STATUS(status);
coorX[0] = strdup("177800");
coorY[0] = strdup("*");
coorValue[0] = NULL;
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("nd1VIA23");
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("*");
coorValue[0] = strdup("8");
coorX[1] = strdup("*");
coorY[1] = strdup("30300");
coorValue[1] = strdup("8");
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
status = defwNetPathVia("nd1VIA23");
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorValue[1]);
status = defwNetPathVia("nd1VIA23");
CHECK_STATUS(status);
coorX[0] = strdup("189560");
coorY[0] = strdup("*");
coorValue[0] = strdup("8");
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
status = defwNetPathVia("nd1VIA12");
```

```
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("27300");
coorValue[0] = strdup("0");
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
status = defwNetPathStart("NEW");
CHECK_STATUS(status);
status = defwNetPathLayer("M3", 1, NULL);
CHECK_STATUS(status);
coorX[0] = strdup("55160");
coorY[0] = strdup("31500");
coorValue[0] = strdup("8");
coorX[1] = strdup("*");
coorY[1] = strdup("34500");
coorValue[1] = strdup("0");
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorValue[1]);
status = defwNetPathVia("M2_M3");
CHECK_STATUS(status);
coorX[0] = strdup("149800");
coorY[0] = strdup("*");
coorValue[0] = strdup("8");
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
status = defwNetPathVia("M2_M3");
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("35700");
```

```
coorValue[0] = NULL;
coorX[1] = strdup("*");
coorY[1] = strdup("37500");
coorValue[1] = NULL;
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
status = defwNetPathVia("M2_M3");
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("*");
coorValue[0] = strdup("8");;
coorX[1] = strdup("170520");
coorY[1] = strdup("*");
coorValue[1] = strdup("0");
status = defwNetPathPoint(2, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
free((char*)coorValue[0]);
free((char*)coorX[1]);
free((char*)coorY[1]);
free((char*)coorValue[1]);
status = defwNetPathVia("M2_M3");
CHECK_STATUS(status);
status = defwNetPathEnd();
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwNet("SCAN");
CHECK_STATUS(status);
status = defwNetConnection("scancell1", "P10", 1);
CHECK_STATUS(status);
status = defwNetConnection("scancell2", "PA0", 1);
CHECK_STATUS(status);
status = defwNetSource("TEST");
CHECK_STATUS(status);
```

```
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwNet("testBug");
CHECK_STATUS(status);
status = defwNetConnection("Z38A05", "G", 0);
CHECK_STATUS(status);
status = defwNetConnection("Z38A03", "G", 0);
CHECK_STATUS(status);
status = defwNetConnection("Z38A01", "G", 0);
CHECK_STATUS(status);
status = defwNetPathStart("ROUTED");
CHECK_STATUS(status);
status = defwNetPathLayer("M1", 0, NULL);
CHECK_STATUS(status);
coorX[0] = strdup("1288210");
coorY[0] = strdup("580930");
coorValue[0] = NULL;
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("GETH1W1W1");
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("582820");
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("GETH2W1W1");
CHECK_STATUS(status);
status = defwNetPathStart("NEW");
CHECK_STATUS(status);
status = defwNetPathLayer("M3", 0, NULL);
CHECK_STATUS(status);
coorX[0] = strdup("1141350");
coorY[0] = strdup("582820");
coorValue[0] = NULL;
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
```

```
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("GETH2W1W1");
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("580930");
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("GETH1W1W1");
CHECK_STATUS(status);
status = defwNetPathStart("NEW");
CHECK_STATUS(status);
status = defwNetPathLayer("M1", 0, NULL);
CHECK_STATUS(status);
coorX[0] = strdup("1278410");
coorY[0] = strdup("275170");
coorValue[0] = NULL;
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathStart("NEW");
CHECK_STATUS(status);
status = defwNetPathLayer("M1", 0, NULL);
CHECK_STATUS(status);
coorX[0] = strdup("1141210");
coorY[0] = strdup("271250");
coorValue[0] = NULL;
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("GETH1W1W1");
CHECK_STATUS(status);
coorX[0] = strdup("*");
coorY[0] = strdup("271460");
coorValue[0] = NULL;
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
```

```
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("GETH2W1W1");
CHECK_STATUS(status);
coorX[0] = strdup("1142820");
coorY[0] = strdup("*");
coorValue[0] = NULL;
status = defwNetPathPoint(1, coorX, coorY, coorValue);
CHECK_STATUS(status);
free((char*)coorX[0]);
free((char*)coorY[0]);
status = defwNetPathVia("GETH3W1W1");
CHECK_STATUS(status);
status = defwNetPathEnd();
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);
free((char*)coorX);
free((char*)coorY);
free((char*)coorValue);

status = defwNet("n1");
CHECK_STATUS(status);
status = defwNetConnection("PIN", "n1", 0);
CHECK_STATUS(status);
status = defwNetConnection("driver1", "in", 0);
CHECK_STATUS(status);
status = defwNetConnection("bumpa1", "bumppin", 0);
CHECK_STATUS(status);
status = defwNetFixedbump();
CHECK_STATUS(status);
status = defwNetEndOneNet();
CHECK_STATUS(status);

status = defwEndNets();
CHECK_STATUS(status);

// SCANCHAIN
status = defwStartScanchains(4);
CHECK_STATUS(status);
status = defwScanchain("the_chain");
```

```
  CHECK_STATUS(status);
  status = defwScanchainCommonscanpins("IN", "PA1", "OUT", "PA2");
  CHECK_STATUS(status);
  status = defwScanchainStart("PIN", "scanpin");
  CHECK_STATUS(status);
  status = defwScanchainStop("cell4", "PA2");
  CHECK_STATUS(status);
  status = defwScanchainOrdered("cell2", "IN", "PA0", NULL, NULL,
                                "cell1", "OUT", "P10", NULL, NULL);
  CHECK_STATUS(status);
  status = defwScanchainFloating("scancell1", "IN", "PA0", NULL, NULL);
  CHECK_STATUS(status);
  status = defwScanchainFloating("scancell2", "OUT", "P10", NULL, NULL);
  CHECK_STATUS(status);
  status = defwScanchain("chain1_clock1");
  CHECK_STATUS(status);
  status = defwScanchainPartition("clock1", -1);
  CHECK_STATUS(status);
  status = defwScanchainStart("block1/current_state_reg_0_QZ", NULL);
  CHECK_STATUS(status);
 status = defwScanchainFloating("block1/pgm_cgm_en_reg", "IN", "SD", "OUT", "QZ");
  CHECK_STATUS(status);
  status = defwScanchainFloating("block1/start_reset_dd_reg", "IN", "SD", "OUT",
"QZ");
  CHECK_STATUS(status);
  status = defwScanchainStop("block1/start_reset_d_reg", NULL);
  CHECK_STATUS(status);
  status = defwScanchain("chain2_clock2");
  CHECK_STATUS(status);
  status = defwScanchainPartition("clock2", 1000);
  CHECK_STATUS(status);
  status = defwScanchainStart("block1/current_state_reg_0_QZ", NULL);
  CHECK_STATUS(status);
  status = defwScanchainFloating("block1/port2_phy_addr_reg_0_", "IN", "SD",
"OUT", "QZ ");
  CHECK_STATUS(status);
  status = defwScanchainFloating("block1/port2_phy_addr_reg_4_", "IN", "SD",
"OUT", "QZ");
  CHECK_STATUS(status);
 status = defwScanchainFloatingBits("block1/port3_intfc", "IN", "SD", "OUT", "QZ",
4);
  CHECK_STATUS(status);
```

```
status = defwScanchainOrderedBits("block1/mux1", "IN", "A", "OUT", "X", 0,
                                  "block1/ff2", "IN", "SD", "OUT", "Q", -1);
CHECK_STATUS(status);
status = defwScanchain("chain4_clock3");
CHECK_STATUS(status);
status = defwScanchainPartition("clock3", -1);
CHECK_STATUS(status);
status = defwScanchainStart("block1/prescaler_IO/lfsr_reg1", NULL);
CHECK_STATUS(status);
status = defwScanchainFloating("block1/dp1_timers", NULL, NULL, NULL, NULL);
CHECK_STATUS(status);
status = defwScanchainFloatingBits("block1/bus8", NULL, NULL, NULL, NULL, 8);
CHECK_STATUS(status);
status = defwScanchainOrderedBits("block1/dsl/ffl", "IN", "SD", "OUT", "Q",
                             -1, "block1/dsl/mux1", "IN", "B", "OUT", "Y", 0);
CHECK_STATUS(status);
status = defwScanchainOrderedBits("block1/dsl/ff2", "IN", "SD", "OUT", "Q",
                             -1, "block1/dsl/mux2", "IN", "B", "OUT", "Y", 0);
CHECK_STATUS(status);
status = defwScanchainStop("block1/start_reset_d_reg", NULL);
CHECK_STATUS(status);

status = defwEndScanchain();
CHECK_STATUS(status);

// GROUPS
groupExpr = (const char**)malloc(sizeof(char*)*2);
status = defwStartGroups(2);
CHECK_STATUS(status);
groupExpr[0] = strdup("cell2");
groupExpr[1] = strdup("cell3");
status = defwGroup("group1", 2, groupExpr);
CHECK_STATUS(status);
free((char*)groupExpr[0]);
free((char*)groupExpr[1]);
status = defwGroupRegion(0, 0, 0, 0, "region1");
CHECK_STATUS(status);
status = defwStringProperty("ggrp", "xx");
CHECK_STATUS(status);
status = defwIntProperty("side", 2);
CHECK_STATUS(status);
```

```
status = defwRealProperty("maxarea", 5.6);
CHECK_STATUS(status);
groupExpr[0] = strdup("cell1");
status = defwGroup("group2", 1, groupExpr);
CHECK_STATUS(status);
free((char*)groupExpr[0]);
status = defwGroupRegion(0, 10, 1000, 1010, NULL);
CHECK_STATUS(status);
status = defwStringProperty("ggrp", "after the fall");
CHECK_STATUS(status);
status = defwGroupSoft("MAXHALFPERIMETER", 4000, "MAXX", 10000, 0, 0);
CHECK_STATUS(status);
status = defwEndGroups();
CHECK_STATUS(status);
free((char*)groupExpr);
status = defwNewLine();
CHECK_STATUS(status);

// BLOCKAGES
xP = (double*)malloc(sizeof(double)*7);
yP = (double*)malloc(sizeof(double)*7);
xP[0] = 2.1;
yP[0] = 2.1;
xP[1] = 3.1;
yP[1] = 3.1;
xP[2] = 4.1;
yP[2] = 4.1;
xP[3] = 5.1;
yP[3] = 5.1;
xP[4] = 6.1;
yP[4] = 6.1;
xP[5] = 7.1;
yP[5] = 7.1;
xP[6] = 8.1;
yP[6] = 8.1;

status = defwStartBlockages(12);
CHECK_STATUS(status);
status = defwBlockageLayer("m1", "comp1");
CHECK_STATUS(status);
status = defwBlockageRect(3456, 4535, 3000, 4000);
```

```
CHECK_STATUS(status);
status = defwBlockageRect(4500, 6500, 5500, 6000);
CHECK_STATUS(status);
status = defwBlockagePolygon(7, xP, yP);
CHECK_STATUS(status);
status = defwBlockagePolygon(6, xP, yP);
CHECK_STATUS(status);
status = defwBlockageRect(5000, 6000, 4000, 5000);
CHECK_STATUS(status);
status = defwBlockagePlacementComponent("m2");
CHECK_STATUS(status);
status = defwBlockageRect(4000, 6000, 8000, 4000);
CHECK_STATUS(status);
status = defwBlockageRect(8000, 400, 600, 800);
CHECK_STATUS(status);
status = defwBlockageLayer("m3", 0);
CHECK_STATUS(status);
status = defwBlockageSpacing(1000);
CHECK_STATUS(status);
status = defwBlockageRect(3000, 4000, 6000, 5000);
CHECK_STATUS(status);
status = defwBlockageLayerSlots("m4");
CHECK_STATUS(status);
status = defwBlockageDesignRuleWidth(1000);
CHECK_STATUS(status);
status = defwBlockageRect(3000, 4000, 6000, 5000);
CHECK_STATUS(status);
status = defwBlockageLayerFills("m5");
CHECK_STATUS(status);
status = defwBlockageRect(3000, 4000, 6000, 5000);
CHECK_STATUS(status);
status = defwBlockageLayerPushdown("m6");
CHECK_STATUS(status);
status = defwBlockageRect(3000, 4000, 6000, 5000);
CHECK_STATUS(status);
status = defwBlockagePolygon(7, xP, yP);
CHECK_STATUS(status);
status = defwBlockagePlacementComponent("m7");
CHECK_STATUS(status);
status = defwBlockageRect(3000, 4000, 6000, 5000);
CHECK_STATUS(status);
```

```
status = defwBlockagePlacementPushdown();
CHECK_STATUS(status);
status = defwBlockageRect(3000, 4000, 6000, 5000);
CHECK_STATUS(status);
status = defwBlockagePlacement();
CHECK_STATUS(status);
status = defwBlockageRect(3000, 4000, 6000, 5000);
CHECK_STATUS(status);
status = defwBlockagePlacementSoft();
CHECK_STATUS(status);
status = defwBlockageRect(4000, 6000, 8000, 4000);
CHECK_STATUS(status);
status = defwBlockagePlacementPartial (1.1);
CHECK_STATUS(status);
status = defwBlockageRect(4000, 6000, 8000, 4000);
CHECK_STATUS(status);
status =  defwBlockageLayerExceptpgnet("metal1");
CHECK_STATUS(status);
status = defwBlockageSpacing(4);
CHECK_STATUS(status);
status = defwBlockagePolygon(3, xP, yP);
CHECK_STATUS(status);
status = defwEndBlockages();
CHECK_STATUS(status);
status = defwNewLine();
CHECK_STATUS(status);
free((char*)xP);
free((char*)yP);

// SLOTS
xP = (double*)malloc(sizeof(double)*7);
yP = (double*)malloc(sizeof(double)*7);
xP[0] = 2.1;
yP[0] = 2.1;
xP[1] = 3.1;
yP[1] = 3.1;
xP[2] = 4.1;
yP[2] = 4.1;
xP[3] = 5.1;
yP[3] = 5.1;
xP[4] = 6.1;
```

```
yP[4] = 6.1;
xP[5] = 7.1;
yP[5] = 7.1;
xP[6] = 8.1;
yP[6] = 8.1;
status = defwStartSlots(2);
CHECK_STATUS(status);
status = defwSlotLayer("MET1");
CHECK_STATUS(status);
status = defwSlotPolygon(7, xP, yP);
CHECK_STATUS(status);
status = defwSlotPolygon(3, xP, yP);
CHECK_STATUS(status);
status = defwSlotRect(1000, 2000, 1500, 4000);
CHECK_STATUS(status);
status = defwSlotRect(2000, 2000, 2500, 4000);
CHECK_STATUS(status);
status = defwSlotRect(3000, 2000, 3500, 4000);
CHECK_STATUS(status);
status = defwSlotLayer("MET2");
CHECK_STATUS(status);
status = defwSlotRect(1000, 2000, 1500, 4000);
CHECK_STATUS(status);
status = defwSlotPolygon(6, xP, yP);
CHECK_STATUS(status);
status = defwEndSlots();
CHECK_STATUS(status);
status = defwNewLine();
CHECK_STATUS(status);
free((char*)xP);
free((char*)yP);

// FILLS
xP = (double*)malloc(sizeof(double)*7);
yP = (double*)malloc(sizeof(double)*7);
xP[0] = 2.1;
yP[0] = 2.1;
xP[1] = 3.1;
yP[1] = 3.1;
xP[2] = 4.1;
yP[2] = 4.1;
```

```
xP[3] = 5.1;
yP[3] = 5.1;
xP[4] = 6.1;
yP[4] = 6.1;
xP[5] = 7.1;
yP[5] = 7.1;
xP[6] = 8.1;
yP[6] = 8.1;
status = defwStartFills(5);
CHECK_STATUS(status);
status = defwFillLayer("MET1");
CHECK_STATUS(status);
status = defwFillRect(1000, 2000, 1500, 4000);
CHECK_STATUS(status);
status = defwFillPolygon(5, xP, yP);
CHECK_STATUS(status);
status = defwFillRect(2000, 2000, 2500, 4000);
CHECK_STATUS(status);
status = defwFillPolygon(7, xP, yP);
CHECK_STATUS(status);
status = defwFillRect(3000, 2000, 3500, 4000);
CHECK_STATUS(status);
status = defwFillLayer("MET2");
CHECK_STATUS(status);
status = defwFillRect(1000, 2000, 1500, 4000);
CHECK_STATUS(status);
status = defwFillRect(1000, 4500, 1500, 6500);
CHECK_STATUS(status);
status = defwFillRect(1000, 7000, 1500, 9000);
CHECK_STATUS(status);
status = defwFillRect(1000, 9500, 1500, 11500);
CHECK_STATUS(status);
status = defwFillPolygon(7, xP, yP);
CHECK_STATUS(status);
status = defwFillPolygon(6, xP, yP);
CHECK_STATUS(status);
status = defwFillLayer("metal1");
CHECK_STATUS(status);
status = defwFillLayerOPC();
CHECK_STATUS(status);
status = defwFillRect(100, 200, 150, 400);
```

```
CHECK_STATUS(status);
status = defwFillRect(300, 200, 350, 400);
CHECK_STATUS(status);
status = defwFillVia("via28");
CHECK_STATUS(status);
status = defwFillViaOPC();
CHECK_STATUS(status);
status = defwFillPoints(1, xP, yP);
CHECK_STATUS(status);
status = defwFillVia("via26");
CHECK_STATUS(status);
status = defwFillPoints(3, xP, yP);
CHECK_STATUS(status);
status = defwEndFills();
CHECK_STATUS(status);
status = defwNewLine();
CHECK_STATUS(status);
free((char*)xP);
free((char*)yP);

// SLOTS
xP = (double*)malloc(sizeof(double)*7);
yP = (double*)malloc(sizeof(double)*7);
xP[0] = 2.1;
yP[0] = 2.1;
xP[1] = 3.1;
yP[1] = 3.1;
xP[2] = 4.1;
yP[2] = 4.1;
xP[3] = 5.1;
yP[3] = 5.1;
xP[4] = 6.1;
yP[4] = 6.1;
xP[5] = 7.1;
yP[5] = 7.1;
xP[6] = 8.1;
yP[6] = 8.1;
status = defwStartSlots(2);
CHECK_STATUS(status);
status = defwSlotLayer("MET1");
CHECK_STATUS(status);
```

```
status = defwSlotRect(1000, 2000, 1500, 4000);
CHECK_STATUS(status);
status = defwSlotPolygon(5, xP, yP);
CHECK_STATUS(status);
status = defwSlotRect(2000, 2000, 2500, 4000);
CHECK_STATUS(status);
status = defwSlotPolygon(7, xP, yP);
CHECK_STATUS(status);
status = defwSlotRect(3000, 2000, 3500, 4000);
CHECK_STATUS(status);
status = defwSlotLayer("MET2");
CHECK_STATUS(status);
status = defwSlotRect(1000, 2000, 1500, 4000);
CHECK_STATUS(status);
status = defwSlotRect(1000, 4500, 1500, 6500);
CHECK_STATUS(status);
status = defwSlotRect(1000, 7000, 1500, 9000);
CHECK_STATUS(status);
status = defwSlotRect(1000, 9500, 1500, 11500);
CHECK_STATUS(status);
status = defwSlotPolygon(7, xP, yP);
CHECK_STATUS(status);
status = defwSlotPolygon(6, xP, yP);
CHECK_STATUS(status);
status = defwEndSlots();
CHECK_STATUS(status);
status = defwNewLine();
CHECK_STATUS(status);
free((char*)xP);
free((char*)yP);

// NONDEFAULTRULES
status = defwStartNonDefaultRules(4);
CHECK_STATUS(status);
status = defwNonDefaultRule("doubleSpaceRule", 1);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal1", 2, 0, 1, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal2", 2, 0, 1, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal3", 2, 0, 1, 0);
```

```
CHECK_STATUS(status);
status = defwNonDefaultRule("lowerResistance", 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal1", 6, 0, 0, 5);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal2", 5, 1, 6, 4);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal3", 5, 0, 0, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleMinCuts("cut12", 2);
CHECK_STATUS(status);
status = defwNonDefaultRuleMinCuts("cut23", 2);
CHECK_STATUS(status);
status = defwNonDefaultRule("myRule", 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal1", 2, 0, 0, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal2", 2, 0, 0, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal3", 2, 0, 0, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleViaRule("myvia12rule");
CHECK_STATUS(status);
status = defwNonDefaultRuleViaRule("myvia23rule");
CHECK_STATUS(status);
status = defwRealProperty("minlength", 50.5);
CHECK_STATUS(status);
status = defwStringProperty("firstName", "Only");
CHECK_STATUS(status);
status = defwIntProperty("idx", 1);
CHECK_STATUS(status);
status = defwNonDefaultRule("myCustomRule", 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal1", 5, 0, 1, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal2", 5, 0, 1, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleLayer("metal3", 5, 0, 1, 0);
CHECK_STATUS(status);
status = defwNonDefaultRuleVia("myvia12_custom1");
CHECK_STATUS(status);
```

```
status = defwNonDefaultRuleVia("myvia12_custom2");
CHECK_STATUS(status);
status = defwNonDefaultRuleVia("myvia23_custom1");
CHECK_STATUS(status);
status = defwNonDefaultRuleVia("myvia23_custom2");
CHECK_STATUS(status);
status = defwEndNonDefaultRules();
CHECK_STATUS(status);
status = defwNewLine();
CHECK_STATUS(status);

// STYLES
status = defwStartStyles(3);
CHECK_STATUS(status);
xP = (double*)malloc(sizeof(double)*6);
yP = (double*)malloc(sizeof(double)*6);
xP[0] = 30;
yP[0] = 10;
xP[1] = 10;
yP[1] = 30;
xP[2] = -10;
yP[2] = 30;
xP[3] = -30;
yP[3] = 10;
xP[4] = -30;
yP[4] = -10;
xP[5] = -10;
yP[5] = -30;
status = defwStyles(1, 6, xP, yP);
CHECK_STATUS(status);
status = defwStyles(2, 5, xP, yP);
CHECK_STATUS(status);
free((char*)xP);
free((char*)yP);
xP = (double*)malloc(sizeof(double)*8);
yP = (double*)malloc(sizeof(double)*8);
xP[0] = 30;
yP[0] = 10;
xP[1] = 10;
yP[1] = 30;
xP[2] = -10;
```

```
yP[2] = 30;
xP[3] = -30;
yP[3] = 10;
xP[4] = -30;
yP[4] = -10;
xP[5] = -10;
yP[5] = -30;
xP[6] = 10;
yP[6] = -30;
xP[7] = 30;
yP[7] = -10;
status = defwStyles(3, 8, xP, yP);
CHECK_STATUS(status);
status = defwEndStyles();
CHECK_STATUS(status);
free((char*)xP);
free((char*)yP);
status = defwNewLine();
CHECK_STATUS(status);

// BEGINEXT
status = defwStartBeginext("tag");
CHECK_STATUS(status);
 defwAddIndent();
status = defwBeginextCreator("CADENCE");
CHECK_STATUS(status);
status = defwBeginextSyntax("OTTER", "furry");
CHECK_STATUS(status);
status = defwStringProperty("arrg", "later");
CHECK_STATUS(status);
status = defwBeginextSyntax("SEAL", "cousin to WALRUS");
CHECK_STATUS(status);
status = defwEndBeginext();
CHECK_STATUS(status);


status = defwEnd();
CHECK_STATUS(status);

lineNumber = defwCurrentLineNumber();
if (lineNumber == 0)
```

```
      fprintf(stderr, "ERROR: nothing has been read.\n");

  fclose(fout);

  return 0;
}
```