

# Layout Database User Manual (*Revision* : 1.12)

*This document expires 28 Feb. 1998*

Kei-Yong Khoo khoo@cs.ucla.edu  
UCLA Computer Science Dept., Los Angeles, CA 90095

September 21, 2000

## 1 Introduction

The layout database (ldb) provides a simple, intuitive and consistent interface to a generic layout database. Different database implementations may be used to support different needs.

### 1.1 Basic Concepts

There are three key concepts in the database: **layout database**, **layout object Id and operations** and **object properties**.

A **layout database** is a hierarchical collections of objects in a common database that describes one or more designs. The database supports file I/O operations to save and retrieve the designs to and from various file formats. The main file format for archiving a design is the GDIF format. The layout database and its operations are described in Section 2.

An object in the database is referenced using a handle called a **layout object Id** or Lid for short. Lid is the only way in which an object in the database can be accessed or manipulated. Therefore, object **operations** such as creation, deletion, searching, copying rely on Lid. The Lid and its operations are described in Section 3.

An object has various **properties** (e.g., name, position) that can be accessed or altered. These operations do not involve the creation or deletion of objects. Different objects have different properties that can be accessed or altered.

### 1.2 Using the database

The database is written in C++ and requires the C++ Standard Template Library that is part of the GNU C++ compiler package (version 2.8.0 and above). You must include the header file “layoutdb.h” in your source file:

```
#include "layoutdb.h"
```

and link in the library file “layoutdb.a” when compiling your program. Under the MCM project, “layoutdb.h” and “layoutdb.a” can be found in the directories  $\$(MCMREL)/include$  and  $\$(MCMREL)/\$(MACH)$ , respectively. Note that the library is available for Solaris (SUN OS 5) only since the C++ Standard Template Library is not installed for SUN OS 4.

## 2 Layout Database

A layout database is a hierarchical collection of objects that describes a design. A layout database is created by declaring a variable of the type `LayoutDatabase`. The contents of the database will be deleted and released from memory when the variable is deleted. The primary format for archiving the database is GDIF. For example, the following code creates a database and fills it with objects from the file “test.gdf.”

```
LayoutDatabase ldb;  
Lid dbId = ldb.read ("test.gdf");
```

Notice that `ldb.read()` returns a `Lid` (layout object id) that can be used to access the database.

The measurement unit in the database is micron scaled by  $\times 1000$ . The values are stored as integers so for instance, 1500 database unit corresponds to 1.5  $\mu\text{m}$ . In future releases, the unit and the scaling factor can be other than micron or 1000, respectively.

### 2.1 Database Operations

1. **read** — read in a file

```
Lid LayoutDatabase::read (char* filename)
```

`read()` opens the file named by *filename* and reads it into the database.

On success, `read()` returns the `Lid` of the database. On failure, it returns a `NULL Lid`.

2. **writeGDIF** — write a GDIF file

```
Lid LayoutDatabase::write (char* filename)
```

`writeGDIF()` opens the file named by *filename* and writes the entire database to the file. If the file exists, then it will be overwritten.

On success, `writeGDIF()` returns 0, otherwise it returns a non-zero error code.

3. **id** — get a `Lid`

```
Lid LayoutDatabase::id ()
```

`id()` returns the `Lid` of the database.

4. **unit** — get the measurement unit (*to-be-implemented*)

```
char* LayoutDatabase::unit ()
```

`unit()` returns a string description of the measurement unit. Commonly used units are *micron*, *inch* and *mil*.

5. **scale** — get the internal scale (*to-be-implemented*)

```
int LayoutDatabase::scale ()
```

All values are stored in the database as integers. A floating point value is converted into an integer by multiplying with the value returned by `scale()`.

6. **getIntParams** — get integer parameters

```
vector<int> LayoutDatabase::getIntParams (char *name)
```

Retrieve a vector of integer parameters.

### 3 Layout Object Id and Operations

Every object in the database must be referenced using a handle called a layout object Id or Lid for short. Besides accessing an object's properties, discussed in Section 4, the Lid supports object operations such such as creation, deletion, searching and copying.

#### 3.1 Layout Object Id (Lid)

An object in the layout database is referenced through a handle called a *layout object Id* or Lid for short. Lid is not unique for an object and so more than one Lids may exist at a time for one object. Deleting an Lid does not delete the underlying object. An Lid becomes invalid when the underlying object is deleted and using an invalid Lid can result in a run-time error.

#### 3.2 Object Operations

Object operations are used to manipulate objects such as insertion and deletion, and for searching or traversing objects in the database.

Objects are created and inserted into a given object by the `insert` member-function with the kind of object as the argument. For example, the following code:

```
// dbId is the id of a database that has been created.
Lid cellId = dbId.insert (Lid::Cell);
Lid portId = cellId.insert (Lid::Port);
```

first creates and inserts a cell into the database (referenced by `dbId`) and then creates and inserts port into the newly created cell. Notice that the newly created cell and port are empty and their information must be filled in.

The first object of a given kind is obtained from a given “parent” object by the `first` member-function with the kind of object as the argument. For example, the following code retrieves the first cell in a database and the first port in that cell:

```
// dbId is the id of a database that has been created.
Lid cellId = dbId.first (Lid::Cell);
Lid portId = cellId.first (Lid::Port);
```

Notice that `dbId` is the object Id of the database and not the database variable itself. Subsequent objects can sequentially access by the *pre*-increment operator. For example, the following codes print out the names of all the ports in a cell:

```
for (Lid portId = cellId.first (Lid::Port); portId; ++portId)
    cout << portId.name () << endl;
```

The list of object operations are summarized below:

1. **insert** — create and insert an object into the database

```
Lid Lid::insert (objectKind kind [, char* name])
```

`insert()` creates an empty object of the specified *kind* and inserts into the object handled by the current Lid. The valid value of *kind* depends on the kind of object that the current Lid is handling. This can be inferred from the objects hierarchy described in Section 5. If the optional *name* is provided, then the name property of the object is set accordingly.

On success, `first()` returns the Lid of the new object, otherwise it returns a NULL Lid.

2. **copy** — duplicate and insert an object into the database (*to-be-implemented*)

```
Lid Lid::copy ()
```

`copy()` duplicates the current object and inserts it into the database.

On success, `first()` returns the Lid of the new object, otherwise it returns a NULL Lid.

3. **erase** — remove an object from the database (*limited-implementation*)

```
void Lid::erase ()
```

`erase()` removes the object handled by the Lid from the database. Currently, the only object that can be removed is the Cell object.

4. **size** — get the number of objects

```
int Lid::size (objectKind kind)
```

The `size()` function returns the number of objects of the specified *kind* contained by the current Lid. The valid value of *kind* depends on the kind of object that the current Lid is handling. This can be inferred from the objects hierarchy described in Section 5.

On success, `size()` returns number of *kind* objects. On failure, `size()` returns 0 and a run-time error message is generated.

5. **first** — search for an specific kind of object

```
Lid Lid::first (objectKind kind [ , char* name])
```

`first()` searches for the first available object contained in *id* of the specified *kind*. If the optional *name* is specified (and not NULL), then the search is narrowed to an object with the same name. The valid value of *kind* depends on the kind of object that the current Lid is handling. This can be inferred from the objects hierarchy described in Section 5.

On success, `first()` returns the Lid of the object found, otherwise it returns a NULL Lid.

6. ++ (**preincrement**) — get the next object

```
Lid Lid::operator++ ()
```

The *pre*-increment operator gets the “next” object of the same kind. All objects in the database can be sequentially accessed by this operator and the first() function.

On success, operator++() returns the Lid of the next object. If no more object is available then a NULL Lid is returned.

7. **integer casting** — test for NULL id

```
int Lid::operator int ()
```

The integer casting operator returns nonzero if the Lid is not NULL (in general, that means it is a valid Lid).

## 4 Object Properties

Every object has a set of properties that can be accessed or altered. While most properties can be read, not all properties can be altered. Some properties cannot be altered because it is derived from other objects. For example, the name of a port reference cannot be changed because it is derived from the name of the instance and the name of the port. Trying to alter a read-only property will result in a run-time error. The list of valid properties for a given object can be determined from the objects hierarchy described in Section 5. For example, the *x* and *y* position of a NetRef (net reference) can only be read.

The property function that accesses or alters a property has the same name as the property as shown in Table 1. Accessing a property requires no argument while setting a property has the new property as the argument. For instance, if instanceId is a valid Lid of an instance, then

```
int x = instanceId.x ();
```

gets the *x* position of the instance and stores it in the variable *x* while

```
instanceId.x (10);
```

moves the *x*-position of the instance to 10.0 (the *y* position remains unchanged). All functions for setting properties return a reference to the Lid so that properties can be set in a single statement:

```
instanceId.x (10).y (20);
```

## 5 Database Objects

The objects defined in the layout database are summarized in Table 1 and described in the following subsections. Each line in the figure begins with the name of the object followed by its properties (italized). An indented line indicates that the object described on that line is contained by the object described on the first previous line with one less level of indentation. For example, a cell object contains ports, instances, paths, nets and texts. The list of valid properties of an object may depend on the value of certain property. In this case, the object may be listed multiple times with different properties. Properties that are read-only are indicated by a *r* superscript.

Table 1: DATABASE OBJECTS

<p>Cell <i>name</i></p> <p>Port <i>name type x y layer</i></p> <p>CellInstance <i>name cellRef x y orient</i></p> <p>Path <i>name</i></p> <p>    PathObj <i>type=Segment x y x1 y1 width layer</i></p> <p>    PathObj <i>type=Via x y width layer</i></p> <p>    PathObj <i>type=String string</i></p> <p>Net <i>name group</i></p> <p>    NetRef <i>type=PortRef name<sup>r</sup> x<sup>r</sup> y<sup>r</sup> portRef</i></p> <p>    NetRef <i>type=PathRef name<sup>r</sup> x<sup>r</sup> y<sup>r</sup> pathRef</i></p> <p>    NetRef <i>type=InstPortRef name<sup>r</sup> x<sup>r</sup> y<sup>r</sup> portRef instRef</i></p> <p>Text <i>name string</i></p> <p>Tech <i>name</i></p> <p>    TechObj <i>name type width</i></p> <p>LayoutRules <i>name</i></p> <p>    LayoutRule <i>type obj obj1 minValue maxValue typValue</i></p>
--

Properties are italicized. A read-only property is indicated by <sup>r</sup>.

## 5.1 Cell

Cell is the basic container for design objects.

- `char* name` — Name of the cell. Every cell must have a name.

## 5.2 Cell:Port

A port is an electrical connection point.

- `char* name` — Name of the port
- `objectKind type` — Type of the port. Valid values are `InputPort`, `OutputPort`, `InOutPort`, `PowerPort`, `GroundPort`.
- `int x, y` — Point location of the port.
- `int layer` — Layer of the port

## 5.3 Cell:CellInstance

An instance of a cell.

- `char* name` — Name of the instance
- `Lid cellRef` — The template (or master) cell of the instance
- `int x, y` — Position of the instance referenced using the (0,0) position of the template cell.

- `int orient` — Orientation of the instance. The orientation is specified as an (scaled by 1000) integer value that indicates the degree of clockwise rotation. A negative value indicates a reflection along the Y-axis before the rotation. For example:
  - 90000 — 90 degrees clockwise rotation
  - -90000 — reflection along the Y-axis followed by a 90 degrees clockwise rotation
  - -360000 — reflection along the Y-axis only

Notice that not all values of orientation are valid and an error message will be printed (and no orientation (i.e., a value of 0) is assumed) if the value is invalid. Currently, only manhattan geometries are supported.

## 5.4 Cell:Path

A path of routing materials most commonly used to represent wires.

- `char* name` — Name of the path

## 5.5 Cell:Path:PathObj

An object in a path.

- `objectKind type` — Type of the path object. The valid values are `Segment`, `Via` and `String`.
- `int width` — Width of the segment or via.
- `int layer` — layer of the segment or via.
- `int x, y` — Position of the object or starting point of a segment.
- `int x1, y1` — Ending point of a segment. This is valid only if `type` is `Segment`.
- `char* string` — A ascii string useful for annotating the objects. This is valid only if `type` is `String`.

## 5.6 Cell:Net

A collection of references to paths and ports to indicate electrical connectivity.

- `char* name` — Name of the net.
- `char* group` — A group name for the net. The usage of the property is not well defined at the moment. It is currently being used as the “global” net name.

## 5.7 Cell:Net:NetRef

An object in a net that referenceing a port.

- `char* name` — (Read only) Name of the net reference.
- `objectKind type` — Type of the net reference. The valid values are `PortRef`, `PathRef` and `InstPortRef`.
- `int x, y` — (Read only) Position of the reference.
- `Lid pathref` — Lid of the path that is being referenced. This is valid only if `type` is `PathRef`.
- `Lid portRef` — Lid of the port that is being referenced. This is valid only if `type` is `PortRef` or `InstPortRef`.
- `Lid instRef` — Lid of the instance (or the instance port) that is being referenced. This is valid only if `type` is `InstPortRef`.

## 5.8 Cell:Text

- `char* name` — Name of the text.
- `char* string` — A null terminated string of the text.

## 5.9 Tech

Tech is a container for technology informations.

- `char* name` — Name of the technology.

## 5.10 Tech:TechObj

A technology object.

- `char* name` — Name of the object.
- `objectKind type` — Type of object. Valid values are `LayerDef` and `ViaDef`.

# 6 Common Parameters

A number of common parameters are used for different phases of the design.

- `number_of_layers` — Number of routing layers. This will be changed to a list of routable layers shortly.
- `wire_spacings` — A list of space separated spacings for each of the layers.
- `via_spacings` — A list of space separated spacings for echo of the via layers. This specification is actually not consistent and will be discontinued.

- `wire_widths` — A list of space separated default widths for each of the layers. Individual nets may requested different widths.
- `via_widths` — A list of space separated default widths for each of the vias. Individual nets may requested different widths.
- `vertical_wire_costs` — A list of space separated cost values for each of the layer when routed vertically. For example, if this is set to “1 2 1 2 1 2,” then vertical is the preferred routing direction for the first layer.
- `horizontal_wire_costs` — A list of space separated cost values for each of the layer when routed horizontally. For example, if this is set to “1 2 1 2 1 2,” then vertical is the preferred routing direction for the first layer.
- `via_costs` — A list of space separated cost values for the vias.

These parameters can be retrieved using the `getIntParams()` function that returns a vector of integers. Notice that the number are scaled so appropriate re-scaling may be necessary. For instance, to get the number of layers:

```
vector<int> v;
v = db.getIntParams ("number_of_layers");
cout << "Number of layers is " << (v[0] / 1000) << endl;
```

## 7 What's New

- A cell object can now be deleted. This has not been tested fully so use it with cautions.
- A new path object has been added. It is a string object. Please see example 4 for usage.
- The Cell object now contains a new class of objects call Text. A Text object has only one property called string.

## 8 Release Schedules

Schedule	Description	Status
Feb.	LayoutDatabase::unit()	
Feb.	LayoutDatabase:: scale()	
Feb.	Implement technology objects	
Mar.	Lid::copy()	
Mar.	Improve the performance of searching in first()	

## 9 Examples

### 9.1 Example 1

The following program prints out the ports and instances in each cell in the entire database.

```
#include "layoutdb.h"

int
main (int argc, char **argv)
{
    if (argc != 2)
    {
        cerr << "Usage: " << argv[0] << " gdif_filename\n";
        return 1;
    }

    LayoutDatabase ldb;
    Lid dbId = ldb.read (argv[1]);        //// Reads in a GDIF file

    for (Lid cellId = dbId.first (Lid::Cell); cellId ; ++cellId)
    {
        cout << "CELL = " << cellId.name () << endl;

        for (Lid portId = cellId.first (Lid::Port); portId; ++portId)
        {
            cout << "    Port = " << portId.name () << " at "
                 << "(" << portId.x () << ", " << portId.y() << ")" << endl;
        }
        for (Lid instId = cellId.first (Lid::CellInstance); instId; ++instId)
        {
            cout << "    Instance = " << instId.name () << " at "
                 << "(" << instId.x () << ", " << instId.y() << ")" << endl;
        }
    }
}
```

### 9.2 Example 2

The following program prints out the points to be routed for used by Patrick's global router:

```
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>

#include "layoutdb.h"

int
main (
    int argc,
    char **argv)
{
    if (argc != 2)
    {
        cerr << "Usage: " << argv[0] << " gdif_filename\n";
    }
}
```

```

    return 1;
}

LayoutDatabase ldb;          //// Creates an empty database
Lid dbId = ldb.read (argv[1]); //// reads in a gdif file
Lid topCell(dbId);          //// To remember the top cell in the DB

//// Assumes that the last cell is the top level cell
for (Lid cell = dbId.first (Lid::Cell); cell; ++cell)
    topCell = cell;

for (Lid net = topCell.first (Lid::Net); net; ++net)
{
    cout << net.size (Lid::NetRef) << endl;

    for (Lid ref = net.first (Lid::NetRef); ref; ++ref)
        cout << ref.x() << " " << ref.y() << " -1 -1\n";
}

return 0;
}

```

### 9.3 Example 3

The following program creates a two-input NAND cell in the database and writes it out in a GDIF file:

```

#include "layoutdb.h"

int
main ()
{
    LayoutDatabase ldb;          //// Creates an empty database
    Lid dbId = ldb.id ();        //// gets the Lid of the database

    Lid cellId = dbId.insert (Lid::Cell, "nand2"); //// Creates a cell

    Lid portId = cellId.insert (Lid::Port, "in0");
    portId.type(Lid::InputPort).x(0).y(0);

    portId = cellId.insert (Lid::Port, "in1");
    portId.type(Lid::InputPort).x(0).y(10000);

    portId = cellId.insert (Lid::Port, "out");
    portId.type(Lid::OutputPort).x(10000).y(5000);

    ldb.writeGDIF ("nand2.gdf");

    return 0;
}

```

### 9.4 Example 4

The following function dumps all the information in the database:

```

int
dump (LayoutDatabase &ldb)
{
    Lid dbId = ldb.id ();          //// gets the Lid of the database

    //// Iterates over all the cells in the database
    for (Lid cell = dbId.first (Lid::Cell); cell; ++cell)
    {
        cout << "CELL " << cell.name () << endl;

        //// Get the first port from the cell and iterates over all the ports
        for (Lid port = cell.first(Lid::Port); port; ++port)
        {
            cout << " " << LayoutObjectKind::name (port.type ())
                << " " << port.name ()
                << " (" << port.x() << ", " << port.y() << ")\n";
        }

        for (Lid inst = cell.first(Lid::CellInstance); inst; ++inst)
        {
            cout << " INSTANCE " << inst.name ()
                << " cell=" << inst.cellRef().name ()
                << " orient=" << inst.orient() << endl;
        }

        for (Lid text = cell.first (Lid::Text); text; ++text)
        {
            cout << " TEXT " << (text.name () ? text.name () : "noname")
                << " string=" << text.string () << endl;
        }

        for (Lid path = cell.first (Lid::Path); path; ++path)
        {
            cout << " PATH " << path.name () << endl;
            for (Lid obj = path.first (Lid::PathObj); obj; ++obj)
            {
                switch (obj.type ())
                {
                    {
                        case Lid::Segment: cout << " SEG "; break;
                        case Lid::Via:     cout << " VIA "; break;
                        case Lid::String:   cout << " STR "; break;
                        default:
                            cerr << "ERROR: Invalid path object!\n";
                            exit (1);
                    }
                }
                if (obj.type () == Lid::Segment || obj.type () == Lid::Via)
                {
                    cout << " W=" << obj.width()
                        << " LEV=" << obj.layer()
                        << " " << obj.x() << " " << obj.y();
                }
                if (obj.type () == Lid::Segment)
                {
                    cout << " " << obj.x1() << " " << obj.y1();
                }
            }
        }
    }
}

```

```

        if (obj.type () == Lid::String)
        {
            cout << " str=" << obj.string();
        }
        cout << endl;
    }
}

for (Lid net = cell.first (Lid::Net); net; ++net)
{
    cout << " NET " << (net.name () ? net.name() : "<noname>")
        << " group=" << (net.group () ? net.group() : "<noname>")
        << endl;

    for (Lid ref = net.first (Lid::NetRef); ref; ++ref)
    {
        if (ref.type () == Lid::PathRef)
        {
            cout << "    Uses path " << ref.pathRef().name() << "\n";
        }
        else if (ref.type () == Lid::PortRef)
        {
            cout << "    portRef: " << ref.portRef().name () << endl;
        }
        else if (ref.type () == Lid::InstPortRef)
        {
            cout << "    instPortRef: "
                << ref.portRef().name ()
                << " inst=" << ref.instRef ().name () << endl;
        }
        else
        {
            cerr << "Invalid type for net reference!\n";
            exit (1);
        }
    }
}
}
return 0;
}

```