A Programmers's Guide to Oct

Rick L. Spickelmier
Peter Moore
A. Richard Newton
Electronics Research Laboratory
Unversity of California, Berkeley
Last Modified: February 28, 1990

August 15, 1990

Contents

1 Introduction **2**

2 The OCT Objects **6**

3 The OCT Functions **17**

4 Staging and Unstaging Facets **37**

5 Accessing OCT Objects **40**

6 Referring to OCT Objects **42**

7 Traversing the OCT Object Graph **43**

8 Instances and Masters **47**

9 Change Propagation and Dealing with Inconsistencies **48**

10 Monitoring Changes **50**

11 Performance **54**

Chapter 1

Introduction

OCT is a data manager for VLSI CAD applications. It is a major component of the Berkeley CAD framework that has been used at Berkeley (and other sites) for the last few years[3, 5]. OCT offers a simple interface for storing information about the various aspects of an evolving chip design. The basic unit in a design is the *cell*. This can be as small as a transistor or NAND gate, or as large as the entire floorplan of a CPU. A cell can consist of instances of other cells, such as a NAND gate consisting of several transistors or the floorplan consisting of an ALU, register file, *etc.*

A cell can have many aspects or *views*, depending on what point in the design you are at and on what design style you use. There can be a schematic view, showing in an abstract way what sub-cells the cell consists of and how they are connected. There can be the symbolic view, where additional information of rough relative placement, sub-cell size and shape, and initial implementation of interconnect might be kept. And even more refined, the physical view, where the implementation is fully defined with exact placement and specific geometry. In addition there are quite different views, such as the simulation view, which might contain the description of the cell in a format that a particular simulator could understand. What views a cell may have, what is contained in those views, and how they are related are not issues addressed by OCT; these decisions are left to authors of the tools that use OCT.

Finally, there is the concept of *facets*. Cells are hierarchical, *i.e.* they contain instances of other cells which in turn may contain instances of other cells, *etc.* For various applications, you may wish to cut off this hierarchy; instead of continuing to traverse the hierarchy by processing the contents of a view, you represent the cell by some simplified abstraction. For a graphics editor, the abstraction might

be a bounding box, for routing the abstraction might be the terminals and routing regions of the cell, and for a design rule-checker it might be just that geometry on the boundary of the cell that needs to be checked against neighboring cells (thus avoiding rechecking the interior of cell each time it is instantiated). We call these various abstractions *facets* of the view.

Each view has a facet named *"contents"* which contains the actual definition of the view, and then various application-dependent interface facets. Again, OCT does not define what interface facets may exist, and what their relation to the contents facet might be. The only facet OCT has any explicit knowledge about is the *contents* facet, which is the facet that defines the name and number of external (formal) terminals a view has. When an instance of a view or a new interface facet is created, the *contents* facet of that view is consulted to find out what terminals the instance or facet inherits from the view. Thus the *contents* facet must be created before any other facet is created and before any instance of that view is created.

The final specification for a facet is the version. The version allows there to be multiple time sequenced facets. Versions are generally ignored by most OCT applications and are only dealt with by data management systems[4].

The facet is the unit that is edited in OCT. A particular facet of a view of a cell is opened and edited independent of the other facets of that view and the other views of that cell. The facet consists of a collection of objects that are related by *attaching* one to another (see Figure 1.1). A box is put on a particular layer by attaching it to that layer, a terminal is shown to be part of a net by attaching it to that net, a box is shown to implement a terminal by attaching it to that terminal. Objects can be attached to more than one object and can have more than one object attached to themselves. Being attached is not particularly strong: if you delete an object, you do not delete objects attached to it, you merely detach the connection. The structure created by attachments can be thought of as a directed graph of OCT objects.

Policy Versus Mechanism

OCT provides a *mechanism* for representing data but places no meaning on the data. *Policy* is used for assigning meaning to the data represented using OCT. For example, OCT has objects that represent layers and geometry, but does not specify how the objects are related to give the *meaning* of a geometry implemented on a given layer. The *policy* states that a geometry that is contained by a layer is implemented on that layer. As another example, OCT has objects that represent nets and terminals, but does not specify how connectivity is represented. The *policy*

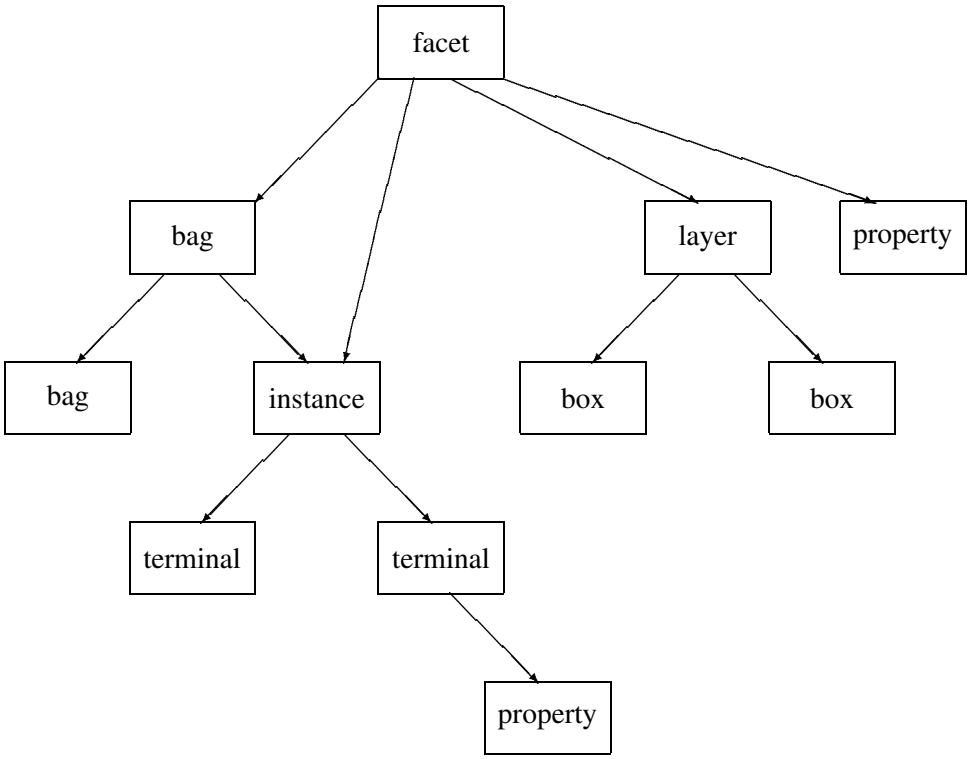

Figure 1.1: Sample Attachments

describes how terminals and nets are used to represented connectivity. See [6] for detailed information about specific OCT policies.

Chapter 2

The OCT Objects

The objects that can be contained in a facet are described by the C language structure *octObject:*

```
struct octObject {
    int type;
    octId objectId;
    union {
        struct octFacet facet;
        struct octInstance instance;
        struct octProp prop;
        struct octTerm term;
        struct octNet net;
        struct octBox box;
        struct octPolygon polygon;
        struct octCircle circle;
        struct octPath path;
        struct octLabel label;
        struct octBag bag;
        struct octLayer layer;
        struct octPoint point;
        struct octEdge Edge;
        struct octChangeList changeList;
        struct octChangeRecord changeRecord;
    } contents;
};
```

type

octObject is used to describe all of the objects contained in the database. The **type** field describes what type of object is contained, and the **contents** field is the union which actually contains the data. Thus if you have an object *obj* that describes a box, *obj.type* has the value `OCT_BOX` and the description of the box is in *obj.contents.box*.

objectId

Along with the data describing the object and the type of the object, each object also contains a field **objectId**. Given the objectId, the function `octGetById` will return the associated object. This means that the objectId can be used as form of pointer to the object, so the entire object need not be kept to reference it: the Id is sufficient. These Ids are guaranteed to be unique across all facets in memory during an OCT session. When an object is deleted, its Id is marked as invalid and may not be used in any further operations; Ids are not reused. Do not make any assumptions about the size of an Id (*i.e.* it may not fit in a `long`).

externalId

As opposed to the *objectId*, which is unique across all objects in all facets currently in memory and persists only as long the the object is in memory, the externalId (xid) persists across sessions. Xids are unique within a facet and combined with the facet that contains the object, the xid is unique across all objects in all facets. The xid is not part of the user-visible OCT object structure, but can be accessed via the function `octExternalId`.

Units

Coordinates in the database, *octCoord*, have no units associated with them. They are simply integers of a reasonable size. It is up to policy to define the units.

OCT_FACET

```
struct octFacet {
    char *cell;
    char *view;
    char *facet;
```

```
    char *version;
    char *mode;
};
```

octFacet describes the facet facet of the view view of the cell cell. The mode field can be one of: "r" (read-only), "w" (create, and delete the old contents), or "a" (append, modify) The version field is not used currently and should be set to OCT_CURRENT_FACET.

OCT_POINT

```
struct octPoint {
    octCoord x;
    octCoord y;
};
```

octPoint describes a point at the coordinates x, y.

OCT_BOX

```
struct octBox {
    struct octPoint lowerLeft;
    struct octPoint upperRight;
};
```

octBox describes a manhattan (all sides parallel to the x or y axes) rectangle. It has its lower left corner (smallest x and y) at the point lowerLeft and upper right corner (largest x and y) at the point upperRight.

OCT_CIRCLE

```
struct octCircle {
    octCoord startingAngle;
    octCoord endingAngle;
    octCoord innerRadius;
    octCoord outerRadius;
    struct octPoint center;
};
```

octCircle actually describes an arc starting at startingAngle (which is in tenths of degrees measured counter-clockwise from the x-axis) and continuing counter-clockwise to endingAngle. The inside edge of the arc is at a distance innerRadius from center, while the outside edge is outerRadius away. OuterRadius should be greater than innerRadius. A simple full circle would have endingAngle equal to 'startingAngle + 3600', outerRadius equal to the radius of the circle, and innerRadius equal to zero.

OCT_PATH

```
struct octPath {
    octCoord width;
};
```

octPath describes a 'wire' of width width. The path followed by the wire is described by an array of points that is set and accessed indirectly, using the functions octGetPoints and octSetPoints.

OCT_POLYGON

```
struct octPolygon {
};
```

octPolygon describes a polygon. The vertices themselves are accessed the same way that the points in octPath are. The first and last vertices are assumed to be connected. The polygon can be self-intersecting, with the interior being those points about which the polygon has non-zero winding number.

OCT_EDGE

```
struct octEdge {
    struct octPoint start;
    struct octPoint end;
};
```

octEdge describes an edge (a zero width line) extending from the point start to the point end. It is included for the use of programs like compactors that may wish to represent constraints between edges of various geometries.

OCT_LABEL

```
struct octLabel {
    char *label;
    struct octBox region;
    octCoord textHeight;
    unsigned vertJust : 2;
    unsigned horizJust : 2;
    unsigned lineJust: 2;
};

OCT_JUST_BOTTOM
OCT_JUST_CENTER
OCT_JUST_TOP
OCT_JUST_LEFT
OCT_JUST_RIGHT
```

The `octLabel` object is used to represent arbitrary text annotation. This text, stored in the field `label`, may be any number of lines each separated by a newline character (`"\n"`). The position of the resulting block of text is given by the box `region`. The height of each line of text is given by the OCT coordinate `textHeight`. The block of text as a whole may be justified within the box `region`. The bit-field `vertJust` specifies the vertical justification of the text block (one of `OCT_JUST_BOTTOM`, `OCT_JUST_CENTER`, or `OCT_JUST_TOP`). The bit-field `horizJust` specifies the horizontal justification of the text block (one of `OCT_JUST_LEFT`, `OCT_JUST_CENTER`, or `OCT_JUST_RIGHT`). Finally, each line of text may be horizontally justified relative to the text block. The bit-field `lineJust` specifies this justification (one of `OCT_JUST_LEFT`, `OCT_JUST_CENTER`, or `OCT_JUST_RIGHT`). Graphically, the layout of the `octLabel` object is shown in Figure 2.1.

OCT_NET

```
struct octNet {
    char *name;
    int32 width;
};
```

`octNet` describes a net with name `name`. The `width` can be used for representing a vector of nets. A net can be used to represent a logical connection

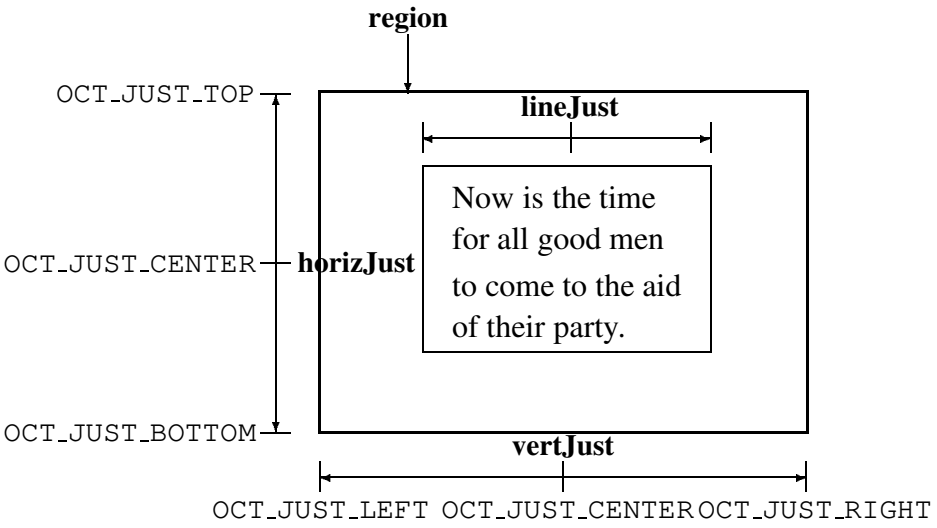

Figure 2.1: The `octLabel` Object

between terminals by attaching the terminals to the net.

OCT_TERM

```
struct octTerm {
    char *name;
    octId instanceId;
    int32 formalExternalId;
    int32 width;
};
```

`octTerm` describes a terminal of an instance or view. If the terminal is a terminal of an instance (an *actual* terminal), then `instanceId` is the `octId` of the instance to which it belongs. If the terminals is a terminal of the current facet (a *formal* terminal), then `instanceId` is equal to `OCT_NULL_ID`. You cannot explicitly create an actual terminal. They are instead implicitly created when an instance is created, with an actual terminal being created and attached to the instance for each formal terminal in the view itself. Formal terminals can only be created in the *contents* facet of a view; all other facets may only reference them.

The width can be used for representing a vector of terminals. In an actual terminal, the formalExternalId is the xid of the formal terminal that the actual terminal represents. The xid of the formal terminal is the same in all facets in the view.

OCT_BAG

```
struct octBag {
    char *name;
};
```

octBag describes an object whose only purpose is to hold other objects. A bag can be used to associate any of the objects in a facet with one another under the name name. As usual, objects are attached and detached from the bag using octAttach and octDetach, while its contents are generated using octInitGenContents and octGenerate. A bag might be used to hold the currently selected items for a graphics editor, the objects on the critical path in a timing simulator, or any other group of objects that should be associated.

OCT_LAYER

```
struct octLayer {
    char *name;
};
```

octLayer describes the layer object for layer name. The major use of the layer object is to hold geometries that are implemented on that layer.

OCT_PROP

```
enum octPropType {
   OCT_NULL, OCT_INTEGER, OCT_REAL, OCT_STRING, OCT_ID,
    OCT_STRANGER, OCT_REAL_ARRAY, OCT_INTEGER_ARRAY
};

struct octProp {
    char *name;
    octPropType type;
```

```
    union octValue {
        int32 integer;
        double real;
        char *string;
        octId id;
        struct {
            int32 length;
            int32 *array;
        } integer_array;
        struct {
            int32 length;
            double *array;
        } real_array;
        struct octUserValue stranger;
    } value;
};

struct octUserValue {
    int length;
    char *contents;
};
```

`octProp` describes the properties that can be attached to any object (including another property). The property is given the name `name` and, depending on the value of `type`, `value` can be an integer, a floating point number, a string, an octId, or a random collection of bytes as defined in `octUserValue`. Because properties can have properties, arbitrarily complex structures can be built up. Note that because properties are no different than any other object, it is possible to attach two properties to an object with the same name.

OCT_INSTANCE

```
struct octInstance {
    struct octTransform transform;
    char *name;
    char *master;
    char *view;
    char *facet;
    char *version;
```

```
};

enum octTransformType {
    OCT_NO_TRANSFORM = 0,        /*  x ->  x, y ->  y */
    OCT_MIRROR_X = 1,            /*  x -> -x, y ->  y */
    OCT_MIRROR_Y = 2,            /*  x ->  x, y -> -y */
    OCT_ROT90 = 3,               /*  x -> -y, y ->  x */
    OCT_ROT180 = 4,              /*  x -> -x, y -> -y */
    OCT_ROT270 = 5,              /*  x ->  y, y -> -x */
    OCT_MX_ROT90 = 6,            /*  x -> -y, y -> -x */
    OCT_MY_ROT90 = 7,            /*  x ->  y, y ->  x */
    OCT_FULL_TRANSFORM = 8
};

struct octTransform {
    struct octPoint translation;
    octTransformType transformType;
    double generalTransform[2][2];
};
```

`octInstance` describes an instance named `name` of the facet `facet` of the view `view` of cell `master` transformed by the transform `transform`. The transform consists of a translation and either 1 of the 8 eight Manhattan transforms defined in `octTransformType` or a general two by two transformation. Instantiating an instance also causes the terminals for that instance to be created as well.

OCT_CHANGE_LIST

```
typedef int32 octObjectMask;
typedef int32 octFunctionMask;

struct octChangeList {
    octObjectMask objectMask;
    octFunctionMask functionMask;
};

OCT_CREATE_MASK
OCT_DELETE_MASK
```

```
OCT_MODIFY_MASK
OCT_PUT_POINTS_MASK
OCT_DETACH_CONTENT_MASK
OCT_DETACH_CONTAINER_MASK
OCT_ATTACH_CONTENT_MASK
OCT_ATTACH_CONTAINER_MASK

OCT_ATTACH_MASK          ('OR' of the ATTACH masks)
OCT_DETACH_MASK          ('OR' of the DETACH masks)

OCT_ALL_FUNCTION_MASK    ('OR' of all masks)
```

`octChangeList` objects are used for monitoring changes in an OCT facet. For each set of operations and objects that is to be monitored, a change list is created in the facet with the `objectMask` field set to the 'OR' of the masks of the object types to be monitored and the `functionMask` field set to the 'OR' of the masks of the operation types to be monitored. For example, the following would monitor all creation and deletion operations on `OCT_BOX` and `OCT_PROP` objects:

```
octObject facet, cl;

cl.type = OCT_CHANGE_LIST;
cl.contents.changeList.objectMask = OCT_BOX_MASK
                                  | OCT_PROP_MASK;
cl.contents.changeList.functionMask = OCT_CREATE
                                    | OCT_DELETE;
octCreate(&facet, &changes);
```

See Section 10 for more information.

OCT_CHANGE_RECORD

```
struct octChangeRecord {
    int32 changeType;
    int32 objectExternalId;
    int32 contentsExternalId;
};
```

`octChangeRecord` objects are created and attached to `octChangeList` objects whenever a monitored change takes place. See Section 10 for more information.

Chapter 3

The OCT Functions

The following describes the OCT functions and gives examples of use for many of them. Many of the OCT functions do not return status codes and many that return status codes raise exceptions for error conditions (via the errtrap package). The exceptions raised are:

Bad objectId: The `objectId` of an object passed to an OCT function is invalid. The `objectId` has: never been assigned, was assigned to an object that has been deleted due to a call to `octDelete`, or released due to a call to `octFreeFacet`. Calls to `octGetById` will not raise an exception if the `objectId` is invalid, but will return `OCT_NOT_FOUND`.

Incorrect Object Type: The type of the object passed to an OCT function is illegal for the function. For example, passing an `OCT_BOX` to `octOpenFacet`.

Internal Inconsistencies: Internal errors in OCT. This type of exception should not be seen.

OCT_NULL_ID

Value of the null OCT objectId.

OCT_CURRENT_VERSION

Value of the facet version field for retrieving the most recent version of a facet.

OCT_MIN_COORD

Value of the smallest `octCoord` allowed.

OCT_MAX_COORD

Value of the largest `octCoord` allowed.

octBegin

```
void
octBegin()
```

Initialize OCT.

octEnd

```
void
octEnd()
```

Release storage associated with OCT.

octOpenFacet

```
octStatus
octOpenFacet(octObject *facet)
```

Bring a facet into memory or create a new facet. If the `mode` field is set to `"r"`, the facet is brought into memory for browsing. Any changes to the data will not persist. If the facet is already in-memory, no new data is brought in. If the facet does not exist, an error is returned (`OCT_ERROR`). If the `mode` field is set to `"a"` and if the facet exists, it is brought into memory for modification. Any changes to the data will persist if the facet is flushed. If the facet is already in-memory and modifiable, no data is read in. If the in-memory facet is not modifiable and the permissions allow modification, the mode is changed. If the in-memory facet is not modifiable and the permissions do not allow modification, an error is returned (`OCT_NO_PERM`). If the facet does not exist, a new one will be created. If the `mode` field is set to `"w"`, a new facet will be created, destroying any existing facet of the

same name. If a facet of the same name is already in-memory, an error is returned (OCT_ERROR). Adding "!" to the mode will force the in-memory facet out of memory (without making the changes persist) and bring in a new copy.

If the facet is new, OCT_NEW_FACET is returned. If the facet is not new but is being brought into memory for the first time, OCT_OLD_FACET is returned. If the facet is already in-memory, OCT_ALREADY_OPEN is returned.

```
facet.type = OCT_FACET;
facet.contents.facet.cell = "cell name";
facet.contents.facet.view = "view name";
facet.contents.facet.facet = "facet name";
facet.contents.facet.version = OCT_CURRENT_VERSION;
facet.contents.facet.mode = "r";
octOpenFacet(&facet);

/* process the data */

octCloseFacet(&facet);
```

octOpenMaster

```
octStatus
octOpenMaster(octObject *instance, *facet)
```

Open the facet facet that corresponds to the instance instance. The cell directory of the facet that contains the instance is searched first, followed by the view directory. The type field of facet should be set to OCT_FACET, the facet field should be set to the particular facet of the view to be opened, and the mode field should be set to either "r" or "a". All other fields in facet are ignored. octOpenMaster returns the same status codes as octOpenFacet.

octOpenRelative

```
octStatus
octOpenRelative(octObject *rfacet, *facet, int location)
```

Open the facet facet. The cell directory of the facet rfacet is searched first, followed by the view directory. All fields in facet must be set. octOpenRelative returns the same status codes as octOpenFacet.

octFlushFacet

```
octStatus
octFlushFacet(octObject *facet)
```

Make the changes to the design data in the facet `facet` persist (if the facet was opened for write or append). No memory is released. Returns `OCT_ERROR` if there is an I/O error.

octCloseFacet

```
octStatus
octCloseFacet(octObject *facet)
```

Make the changes to the design data in the facet `facet` persist (if the facet was opened for write or append) and release all of the space associated with the facet `facet`. If the facet has been opened multiple times, the facet will have to be closed multiple times to release the memory (a reference count is kept), for example:

```
while (octGetById(facet) == OCT_OK) {
    octCloseFacet(facet);
}
```

`octCloseFacet` is equivalent to:

```
octFlushFacet(facet);
octFreeFacet(facet);
```

Note that that each call to `octCloseFacet` updates the persistent copy. The better way to update the persistent copy and release the memory is:

```
octCloseFacet(facet);
while (octGetById(facet) == OCT_OK) {
    octFreeFacet(facet);
}
```

Returns the error codes associated with `octFlushFacet` and `octFreeFacet`.

octWriteFacet

```
octStatus
octWriteFacet(octObject *new, *old)
```

Write the in-memory facet `old` to the facet specified by `new`. The facet `old` is kept in-memory. The facet `new` is not in-memory at the end of the call.

Returns `OCT_ERROR` if there is an I/O error.

octCopyFacet

```
octStatus
octCopyFacet(octObject *new, *old)
```

Create a new persistent version of the facet using the `cell`, `view`, `facet`, and `version` specification in `new` with the same contents as the current persistent copy or in-memory copy of the facet `old`. `new` should not already be open and `octCopyFacet` does not open it. If `old` is open the in-memory copy of `old` will be used; if it is not open, the persistent version will be used.

Returns the error codes associated with `octOpenFacet` and `octWriteFacet`.

octFreeFacet

```
octStatus
octFreeFacet(octObject *facet)
```

Release the space associated with the facet `facet`. If the facet has been opened multiple times, the facet will have to be freed multiple times (a reference count is kept and only when the count reaches zero is the memory actually released), for example:

```
while (octGetById(facet) == OCT_OK) {
    octFreeFacet(facet);
}
```

See also `octCloseFacet`.

octGetFacetInfo

```
octStatus
octGetFacetInfo(octObject *facet, struct octFacetInfo *info)

struct octFacetInfo {
    int32 timeStamp;
    int32 bbDate;
    int32 formalDate;
    int32 createDate;
    int32 deleteDate;
    int32 modifyDate;
    int32 attachDate;
    int32 detachDate;
    int numObjects;
    struct octBox bbox;
    char *fullName;
};
```

Returns in `info` the facet info structure for the facet `facet`. The facet info structure consists of timestamps of the last time various OCT operations were performed, the facet bounding box, the number of objects in the facet, and the full name of the facet. The timestamps are not actually times, but rather the value of the `timestamp` in the info structure when the operation was done. `timestamp` is in turn incremented whenever any operation is done. So if the value of `timestamp` is 2645 and an `octCreate` is done on the facet, then both `timestamp` and `createDate` are set to 2646.

octFullName

```
void
octFullName(octObject *facet, char **name)
```

Returns in `name` the full (*absolute*) cell name of the facet. This is useful for fully qualifying a design (*i.e.* removing the relative names). Space is allocated for `name` and the user is responsible for releasing the memory.

octCreate

```
octStatus
octCreate(octObject *cnt, *obj)
```

Creates the object `obj` attached to object `cnt`. This equivalent to creating an object without any attachments and then calling `octAttach(cnt, obj)`. Same return values as `octAttach`.

octDelete

```
octStatus
octDelete(octObject *obj)
```

Removes the object `obj` from its associated facet and detaches it from all of its containers. No further references can be made to this object.

octModify

```
octStatus
octModify(octObject *obj)
```

Modify the object `obj`. Not every field in an object is modifiable. The fields that are allowed to change are:

Type	field(s)
OCT_INSTANCE	name, transform
OCT_PROP	name, value
OCT_LAYER	name
OCT_TERM	name
OCT_NET	name
OCT_BAG	name
OCT_FACET	NOTHING
other object	all fields

For example:

```
octObject facet, layer;

layer.type = OCT_LAYER;
layer.contents.layer.name = "POLY";
if (octGetByName(&facet, &layer) == OCT_OK) {
    layer.contents.layer.name = "POLY2";
    octModify(&layer);
}
```

octAttach

```
octStatus
octAttach(octObject *cnt, *obj)
```

Attach object *obj* to the object *cnt*. The result of this operation is that *cnt contains obj* and *obj* is *contained* by *cnt*. Objects may be attached to the same container multiple times. OCT_OK is returned for success. OCT_ERROR is returned if the objects are in different facets or *cnt* is not allowed to contain objects of the type of *obj*.

octAttachOnce

```
octStatus
octAttachOnce(octObject *cnt, *obj)
```

If obj is not already contained by cnt, attach obj to cnt. Functionally the same as:

```
if (octIsAttached(cnt, obj) == OCT_NOT_ATTACHED) {
    octAttach(cnt, obj);
}
```

OCT_OK is returned for success, OCT_ALREADY_ATTACHED if obj is already contained by cnt, and OCT_ERROR if the objects are in different facets.

octIsAttached

```
octStatus
octIsAttached(octObject *cnt, *obj)
```

Returns OCT_OK if cnt contains obj, OCT_NOT_ATTACHED if cnt does not contain obj, and OCT_ERROR if the objects are in different facets.

octDetach

```
octStatus
octDetach(octObject *cnt, *obj)
```

Removes the attachment between cnt and obj; cnt no longer contains obj. If there are multiple attachments between the objects, octDetach must be called to remove each one, for example:

```
while (octIsAttached(cnt, obj) == OCT_OK) {
    octDetach(cnt, obj);
}
```

OCT_OK is returned for success. OCT_NOT_ATTACHED is returned if the objects are not attached. OCT_ERROR is returned if the objects are in different facets.

octPutPoints

```
octStatus
octPutPoints(octObject *obj, int32 num, octPoint *pnts)
```

Associate the array of points in pnts with the object obj. num contains the number of points in the array. Only polygons and paths have points associated with them. Returns OCT_ERROR if the type of obj can not have points associated with it.

```
octObject facet, path;
octPoint points[2];

path.type = OCT_PATH;
path.contents.path.width = 40;
octCreate(&facet, &path);
points[0].x = 0;
points[0].y = 0;
points[1].x = 100;
points[1].y = 0;
octPutPoints(&path, 2, points);
```

octGetPoints

```
octStatus
octGetPoints(octObject *obj, int32 *num, octPoint *pnts)
```

Returns in `pnts` the points associated with the object `obj`. Only polygons and paths have points associated with them. `*num` contains the size of the user supplied point array `pnts`. If the array is too small, `OCT_TOO_SMALL` is returned. In any case, on return `*num` will contain the number of points associated with the object. Returns `OCT_ERROR` if the type of `obj` can not have points associated with it. Example use:

```
    int32 num = 0;
    octPoint *points;
    octObject object;

  if (octGetPoints(&object, &num, points) == OCT_TOO_SMALL) {
       points = ALLOC(octPoint, num);
       octGetPoints(&object, &num, points);
    }
```

octInitGenContents

```
octStatus
octInitGenContents(octObject *cnt, octObjectMask mask,
                   octGenerator *gen)
```

Create a generator `gen` that will return all objects of type(s) `mask` contained by the object `obj`. `mask` is an 'OR'ing of the following:

OCT_FACET_MASK	OCT_TERM_MASK
OCT_NET_MASK	OCT_INSTANCE_MASK
OCT_PROP_MASK	OCT_BAG_MASK
OCT_POLYGON_MASK	OCT_BOX_MASK
OCT_CIRCLE_MASK	OCT_PATH_MASK
OCT_LABEL_MASK	OCT_LAYER_MASK
OCT_POINT_MASK	OCT_EDGE_MASK
OCT_FORMAL_MASK	OCT_CHANGE_LIST_MASK
OCT_CHANGE_RECORD_MASK	OCT_FORMAL_MASK

For example:

```
OCT_INSTANCE_MASK | OCT_BOX_MASK | OCT_BAG_MASK
```

OCT_GEO_MASK is an 'OR'ing of all geometric object types and OCT_ALL_MASK is an 'OR'ing of all object types.

octInitGenContainers

```
octStatus
octInitGenContainers(octObject *obj, octObjectMask mask,
                     octGenerator *gen)
```

Create a generator gen that will return all objects of type(s) mask that contain the object obj.

octGenerate

```
octStatus
octGenerate(octGenerator *gen, octObject *obj)
```

Return in the object obj the next object in the generator sequence specified by the generator gen. Returns OCT_OK for success. Returns OCT_GEN_DONE when the sequence is exhausted.

Note that generation loops should check for OCT_OK, not OCT_GEN_DONE. If an error occurs and OCT_ERROR is returned, checking for not equal to OCT_GEN_DONE will cause an infinite loop.

```
/* proper way to generate */
while (octGenerate(&gen, &obj) == OCT_OK) {
    /* do something */
}

/* XXX wrong way to generate */
while (octGenerate(&gen, &obj) != OCT_GEN_DONE) {
    /* do something */
}
```

octFreeGenerator

```
octStatus
octFreeGenerator(octGenerator *gen)
```

Free the storage associated with the generator `gen`. Storage is automatically released by OCT when a generator runs to completion (returns `OCT_GEN_DONE`), however `octFreeGenerator` can be called safely at the end of all generation sequences.

```
while (octGenerate(&gen, &obj) == OCT_OK) {
    /* do something */
}
octFreeGenerator(&gen);
```

octGenFirstContent

```
octStatus
octGenFirstContent(octObject *obj, octObjectMask mask,
                   octObject *result)
```

Same as:

```
octGenerator gen;
octStatus status;

octInitGenContents(obj, mask, &gen);
status = octGenerate(&gen, result);
octFreeGenerator(&gen);
```

octGenFirstContainer

```
octStatus
octGenFirstContainer(octObject *obj, octObjectMask mask,
                     octObject *result)
```

Same as:

```
    octGenerator gen;
    octStatus status;

    octInitGenContainers(obj, mask, &gen);
    status = octGenerate(&gen, result);
    octFreeGenerator(&gen);
```

octInitUserGen

```
octStatus
octInitUserGen(char *state, octStatus (*gen_func)(),
           octStatus (*free_func)(), octGenerator *gen)
```

Create a generator based on a user's generation function.

```
octStatus
gen_func(state_p, obj)
char **state_p;
octObject *obj;
```

`Gen_func` is called by `octGenerate` each time a new object is needed. `state_p` is a pointer to the state given in `octInitUserGen`. `obj` is the object to be filled in. If you wish to explicitly to free all the storage associated with generator (for example after generating the last object), call `octFreeGenerator` on `gen`. This allows OCT to free the state it allocated.

```
octStatus
free_func(state)
char **state_p;
```

`Free_func` is called by `octFreeGenerator` to free all state associated with the user generator.

For example, the function `init_skipping_gen` creates a generator that returns every other object, rather than every object, when `octGenerate` is called.

```
octStatus
skip_free(state)
char *state;
{
```

```
   int retval = octFreeGenerator((octGenerator *) state);
    (void) free(state);
    return retval;
}

octStatus
skip_gen(state_p, obj)
char **state_p;
octObject *obj;
{
    octObject junk;
    octStatus retval;

    octGenerator *my_gen = (octGenerator *) *state_p;
    retval =  octGenerate(my_gen, &junk);
    if (retval != OCT_OK) {
        return retval;
    }
    return octGenerate(my_gen, obj);
}

octStatus
init_skipping_gen(container, mask, gen_p)
octObject *container;
octObjectMask mask;
octGenerator *gen_p;
{
    octGenerator *my_gen = ALLOC(octGenerator);
    int retval;

    if (my_gen == NIL(octGenerator)) {
        return OCT_ERROR;
    }

   retval = octInitGenContents(container, mask, my_gen);

    if (retval != OCT_OK) {
        return retval;
    }
```

```
    return octInitUserGen((char *) my_gen, skip_gen,
                          skip_free, gen_p);
}
```

octBB

```
octStatus
octBB(octObject *obj, octBox *bbox)
```

Returns in `bbox` the bounding box of the object `obj`. Returns `OCT_OK` for success, `OCT_NO_BB` if the object does not have a bounding box. OCT defines bounding boxes for geometries, instances, and facets.

octGetFacet

```
void
octGetFacet(octObject *obj, octObject *facet);
```

Returns in `facet` the facet that contains the object `obj`.

octGetByName

```
octStatus
octGetByName(octObject *cnt, *obj)
```

Returns in `obj` the object of type `obj->type` with the name specified in `obj` that is contained by `cnt`. Only certain objects can have names (term, net, instance, prop, layer, bag, and label). If there is more than one object that matches, the one that is returned is undefined. If none match, `OCT_NOT_FOUND` is returned. If the type of the object `obj` does not have a name, `OCT_ERROR` is returned. Example use:

```
    octObject facet, object;

    object.type = OCT_LAYER;
    object.contents.layer.name = "POLY";
    octGetByName(&facet, &object);
```

octContainerGetByName

```
octStatus
octGetContainerByName(octObject *obj, *cnt)
```

Returns the object of type `cnt->type` with the name specified in `cnt` that contains `obj`. Only certain objects can have names (term, net, instance, prop, layer, bag, and label). If there is more than one object that matches, the one that is returned is undefined. If none match, `OCT_NOT_FOUND` is returned. If the type of the object `cnt` does not have a name, `OCT_ERROR` is returned.

octGetById

```
octStatus
octGetById(octObject *obj)
```

Returns in `obj` the object with the same objectId as `obj->objectId`. Returns `OCT_NOT_FOUND` if no object has the specified objectId. The following is an example of the use of `octGetById`:

```
octObject object;
octId objId;

object.objectId = objId;
octGetById(&object);
```

octExternalId

```
void
octExternalId(octObject *obj, int32 *xid)
```

Returns in `xid` the externalId for the object `obj`.

octGetByExternalId

```
octStatus
octGetByExternalId(octObject *facet, int32 xid,
                   octObject *obj)
```

Returns in `obj` the object with externalId `xid` in the facet `facet`. Returns `OCT_NOT_FOUND` if no object exists in facet `facet` with externalId `xid`.

octFormatId

char * octFormatId(id) octId id;

Return a pointer to a static buffer (don't free it) that contains a printable representation of an Oct id.

octIdsEqual

```
int
octIdsEqual(octId id1, octId id2)
```

Returns 1 if the `objectId's` are equal, 0 if they are not.

octIdIsNull

```
int
octIdIsNull(octId id)
```

Returns 1 if the `objectId` is null, 0 if it is not.

octMarkTemporary

```
void
octMarkTemporary(octObject *obj)
```

Mark the object `obj` as temporary. Temporary objects are never allowed to persist, so they are effectively deleted once you close the facet.

octUnmarkTemporary

```
void
octUnmarkTemporary(octObject *obj)
```

Mark the object `obj` as permanent. The object will now be allowed to persist on closing the facet.

octIsTemporary

```
int
octIsTemporary(octObject *obj)
```

Returns 1 if the object `obj` is temporary, 0 if it is not.

octTransformGeo

```
void
octTransformGeo(octObject *obj, octTransform *xform)
```

Apply the transform `xform` to the object `object`, changing it in place. The actual object in 'OCT space' is not changed; only your copy is changed. Use `octTransformAndModifyGeo` to transform and modify the object.

octTransformAndModifyGeo

```
octStatus
octTransformAndModifyGeo(octObject *obj,
                         octTransform *xform)
```

Same as:

```
octTransformGeo(obj, xform);
octModify(obj);
```

octTransformPoints

```
void
octTransformPoints(int32 num, octPoint *pnts,
                   octTransform *xform)
```

Apply the transform `xform` to the array of points `pnts`. The array is expected to have `num` points. `octPutPoints` must be called to update the database.

octScaleGeo

```
void
octScaleGeo(octObject *obj, double scale)
```

Scale the object `object` by the scale factor `scale`, changing it in place. The actual object in 'OCT space' is not changed; only your copy is changed. Use `octScaleAndModifyGeo` to transform and modify the object.

octScaleAndModifyGeo

```
octStatus
octScaleAndModifyGeo(octObject *obj, double scale)
```

Same as:

```
octScaleGeo(obj, scale);
octModify(obj);
```

octCreateOrModify

```
octStatus
octCreateOrModify(octObject *cnt, *obj)
```

Same as:

```
octObject dummy;
dummy = *obj;
if (octGetByName(cnt, &dummy) == OCT_NOT_FOUND) {
    octCreate(cnt, obj);
} else {
    obj->objectId = dummy.objectId;
    octModify(obj);
}
```

octGetOrCreate

```
octStatus
octGetOrCreate(octObject *cnt, *obj)
```

Same as:

```
if (octGetByName(cnt, obj) ==  OCT_NOT_FOUND) {
    octCreate(cnt, obj);
}
```

octErrorString

```
char *
octErrorString()
```

Returns a pointer to a textual description of the last error.

octError

```
void
octError(string)
```

Print to standard error a description of the last error. This is equivalent to:

```
fprintf(stderr, "%s: %s\n", string, octErrorString());
```

Chapter 4

Staging and Unstaging Facets

Staging is the process of bringing the persistent design data contained in a facet into memory. Unstaging is the process of making the modifications to the design data in a facet persist (if requested) and releasing the resources associated with the in-memory design data in the facet. The functions `octOpenFacet`, `octOpenMaster`, and `octOpenRelative` bring the contents of a facet into memory. Since there is currently no paging support, the sum total of the memory required for all opened facets must be less than the per process virtual memory limit[1]. These functions can return multiple values:

OCT_OLD_FACET: A persistent copy of the facet exists and has been read in for the first time this session.

OCT_NEW_FACET: There was no persistent copy of the facet, no in-memory copy existed, and a new in-memory copy has been created.

OCT_ALREADY_OPEN: The facet is already in-memory. There may or may not also be a persistent version.

OCT_ERROR: An error condition. The facet may be corrupt, or there may not be a persistent version and an open for read was requested.

OCT_NO_PERM: The program does not have permission to read, modify, or create the persistent version of the facet.

OCT_INCONSISENT: A persistent version of the facet exists and has been read in, however some of the masters have not been found or terminals

[1] Information on buffering and clustering objects in OCT can be found in [1].

have been changed and need manual cleanup. Generating on the bag `OCT_INCONSISTENT_BAG` will return the masters/terminals that are inconsistent.

The three facet opening functions, `octOpenFacet`, `octOpenMaster`, and `octOpenRelative`, have different methods for locating the persistent copies of the facet[2]. If the cell name of the facet is fully qualified (*absolute*), which means that no context-dependent information is necessary to find the facet, then the facet can be directly accessed. If the name is not fully qualified (*relative*), which means that context dependent information is used, then the three routines behave in the following manner. For `octOpenFacet`, the current directory is searched. For `octOpenMaster`, the cell and view directories of the facet that contain the instance of the master are searched. For `octOpenRelative`, the cell and view directories of the facet that is used as the relative facet is searched. Using the cell directory creates a flat hierarchy on the file system and using the view directory maps the instantiation hierarchy onto the hierarchy of the file system.

The function `octFlushFacet` makes the facet persist[3] without releasing the memory associated with the facet. The function `octFreeFacet` releases the memory associated with the facet, without making the changes persist, if the reference count is zero. Each open of a facet increments a reference count and each free/close decrements the reference count; when the count reaches zero the memory is released. The function `octCloseFacet` combines `octFlushFacet` and `octFreeFacet`.

Information Records

Occasionally just the basic information about a facet is needed: the number objects, the bounding box, the last time certain operations were performed, *etc.* The function `octGetFacetInfo` can be called to bring in only this information.

```
struct octFacetInfo {
```

[2]The terminology used in this section along with the details of facet location are based upon the basic UNIX file system representation of OCT facets. In this representation cells are represented as directories, views are directories inside of cell directories, and facets are files inside of view directories. Two data management systems have been written for OCT that do not store and locate facets in exactly the same manner as described in this section.

[3]To make a facet persist means that the design data in the facet can be accessed by new tool invocations. In the current OCT, each facet is represented as a file and making a facet persist means writing the data to the facet file.

```
    int32 timeStamp;
    int32 bbDate;
    int32 formalDate;
    int32 createDate;
    int32 deleteDate;
    int32 modifyDate;
    int32 attachDate;
    int32 detachDate;
    int numObjects;
    struct octBox bbox;
    char *fullName;
};
```

Problems

You may not be able to directly get the information records for OCT facets retrieved by `octOpenMaster` and `octOpenRelative`. The function used to retrieve information records, `octGetFacetInfo`, goes to persistent storage to find the record and uses the same resolve mechanism for facet location as `octOpenFacet`. The following is guaranteed to work (assuming the facet actually resides in persistent storage):

```
    octObject facet;
    struct octFacetInfo info;
    char *name;

    octFullName(&facet, &name);
    facet.contents.facet.cell = name;
    octGetFacetInfo(&facet, &info);
```

Chapter 5

Accessing OCT Objects

OCT functions do not give the user access to the actual data that was brought into memory (referred to as 'OCT space') due to an `octOpenFacet`, `octOpenMaster`, or `octOpenRelative` call, but instead they copy the user-visible portion of the OCT object into a user-supplied structure. This provides a layer of safety: the user can modify and destroy his copy of the object without corrupting the data in OCT space. A commit operation (such as `octModify`) is necessary to update OCT space. The only exception to this is in the case of strings. The char pointers for strings (bag names, property values, *etc.*) are pointers into the data in OCT space. Thus directly modifying or freeing the string is not allowed and will corrupt 'OCT space'.

```
octObject facet, object;

object.type = OCT_BAG;
object.contents.bag.name = save_string("bob");
octGetByName(&facet, &object);
free(object.contents.bag.name);  /* XXX illegal */
```

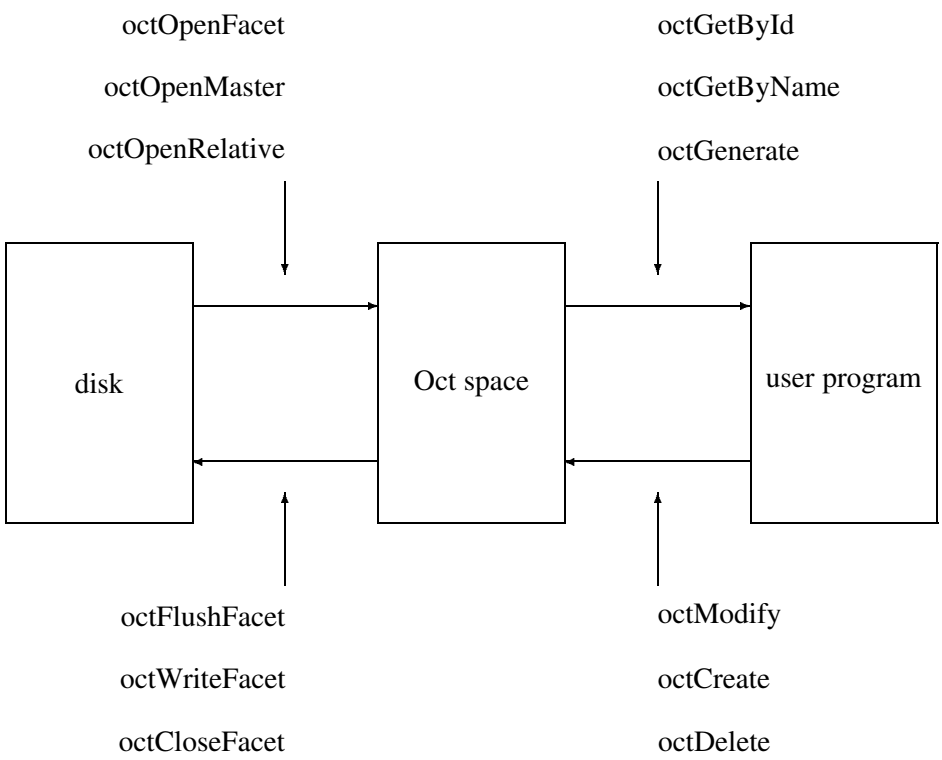

Figure 5.1: OCT Operations and the Environment

Chapter 6

Referring to OCT Objects

All objects have two methods for directly accessing them and some have a third method. There are two identifiers associated with each object. The first is a per-session object identifier (objectId) that is unique across all facets that are currently in memory. The objectId is part of the OCT object structure and `octGetById` will retrieve the object. The second identifier is called an external identifier (xid). This identifier is unique inside of a facet, and combined with the facet is unique across all facets in and out of memory. This identifier persists across sessions and can be used for relating objects across facets. The xid is not part of the user visible OCT structure and must be accessed by the OCT function `octExternalId`. The OCT function `octGetByExternalId` is used to retrieve an object based on its xid.

```
octGetById(&object);

octGetByExternalId(&facet, xid, &object);

object.type = OCT_BAG;
octGetByName(&facet, &object);
```

The third method of retrieving an object based on an identifier is by name. Some object types have names associated with them (properities, bags, instances, layers, terminals, nets, and labels). The call `octGetByName` takes a name, type, and container and returns the object of the given name and type contained by the container. If multiple objects of the same type and container have the same name, the particular object that is returned is not defined; OCT does not try to force uniqueness of names.

Chapter 7

Traversing the OCT Object Graph

There are two ways to traverse the object graph. One is by name/Id reference as described previously (`octGetById`, `octGetByName`, *etc.*). The other is by using `generators`. Generators are used for sequencing through objects of a given type or set of types that are attached to an object (such as all terminals attached to a net). Generators must be *initialized* by a call to `octInitGenContents` or `octInitGenContainers`, and objects are returned in sequence by subsequent calls to `octGenerate`. For example:

```
/* generate all nets attached to the facet */
octInitGenContents(&facet, OCT_NET_MASK, &gen);
while (octGenerate(&gen, &net) == OCT_OK) {
    /* do something */
}
```

If objects that would occur in the generation sequence are modified such that they will not be in the generation sequence (deleted, detached from the container), they will be skipped. If new objects appear (creation or attachment to the container) they will be seen by the generator. Note that this means the first example below will finish and the next will loop forever:

```
/* safe generation sequence */
octInitGenContents(&facet, OCT_NET_MASK, &gen);
while (octGenerate(&gen, &net) == OCT_OK) {
    octDelete(&net);
}
```

```
    /* XXX will loop forever */
    newnet.type = OCT_NET;
    newnet.contents.net.name = "new net";
    octInitGenContents(&facet, OCT_NET_MASK, &gen);
    while (octGenerate(&gen, &net) == OCT_OK) {
        octCreate(&facet, &newnet);
    }
```

The following two examples show code for real OCT applications: generating all of the geometry in a design, and producing a netlist. Both of these go beyond what OCT defines and use *policy* specific information to determine what attachments to follow.

```
/*
 * program to generate over all geometries in the facet
 */

#include "copyright.h"
#include "port.h"
#include "oct.h"

main(argc, argv)
int argc;
char **argv;
{
    /* declare the oct objects to be used */
    octObject facet;          /* facet to be opened            */
    octObject layer;          /* layer containing the geometry */
    octObject geo;            /* geometry on the layer         */

    /* declare the oct generators to be used */
    octGenerator lgen;        /* generator for the layers      */
    octGenerator ggen;        /* generator for the geometries  */

   /* initialize oct - alloc. tables, notify design managers, etc. */
    octBegin();

    /* open the facet */
    facet.type = OCT_FACET;
    facet.contents.facet.cell = argv[1];
    facet.contents.facet.view = argv[2];
    facet.contents.facet.facet = "contents";
    facet.contents.facet.version = OCT_CURRENT_VERSION;
    facet.contents.facet.mode = "r";
    if (octOpenFacet(&facet) < OCT_OK) {
```

```
        octError("opening facet to be generated");
        exit(-1);
    }

    /* generate over all layers */

    (void) octInitGenContents(&facet, OCT_LAYER_MASK, &lgen);

    while (octGenerate(&igen, &layer) == OCT_OK) {

        /* generate over all geometries on the layer */

        (void) octInitGenContents(&layer, OCT_GEO_MASK, &ggen);

        while (octGenerate(&ggen, &geo) == OCT_OK) {

            /* process the geometry */
        }
    }

   /* close down oct - release memory, notify design managers, etc. */
    octEnd();

    exit(0);
}

/*
 * This is a simple program that netlists a facet
 */

#include "copyright.h"
#include "port.h"
#include "oct.h"

main(argc, argv)
int argc;
char **argv;
{
    octObject facet, instance, term, net;
    octGenerator igen, tgen;

    /* initialize oct */
    octBegin();

    /* open the facet to netlist */
    facet.type = OCT_FACET;
    facet.contents.facet.cell = argv[1];
    facet.contents.facet.view = argv[2];
```

```
    facet.contents.facet.facet = "contents";
    facet.contents.facet.version = OCT_CURRENT_VERSION;
    facet.contents.facet.mode = "r";
    if (octOpenFacet(&facet) < OCT_OK) {
        octError("opening facet to be netlisted");
        exit(-1);
    }

    /* generate over all instances */

    (void) octInitGenContents(&facet, OCT_INSTANCE_MASK, &igen);

    while (octGenerate(&igen, &instance) == OCT_OK) {

        (void) printf("%s(%s)", instance.contents.instance.name,
                    instance.contents.instance.master);

        /* generate over all terminals on the instance */

        (void) octInitGenContents(&instance, OCT_TERM_MASK, &tgen);

        while (octGenerate(&tgen, &term) == OCT_OK) {

            /* find the net that contains the terminal */
        if (octGenFirstContainer(&term, OCT_NET_MASK, &net) != OCT_OK) {
                /* not connected to any net */
                (void) printf(" %s=<unconnected>",
                            term.contents.term.name);
            } else {
                (void) printf(" %s=%s", term.contents.term.name,
                            net.contents.net.name);
            }
        }
    }

    /* close down oct */
    octEnd();

    exit(0);
}
```

Chapter 8

Instances and Masters

OCT allows the specification of hierarchy by providing instance objects. Instance object, OCT objects of type `OCT_INSTANCE`, are used for referring to other OCT facets. The OCT facet referred to by the instance is known as the *master* of the instance.

There are two ways to reference facets using instances: *relative* and *absolute*. Relative means that the context (location) of the facet that contains the instance is used to find the master of the instance. Relative allows the entire design to be moved without modifying the instance objects. Absolute means that the context is not used to find the master of the instance; all the necessary information is contained in the instance object.

When an instance of a master is created, a copy of the formal terminal of the master, called an actual terminal is created for each formal terminal in the master (and attached to the instance).

Chapter 9

Change Propagation and Dealing with Inconsistencies

One difference between a static data format (*i.e.* EDIF) and a database and interface (*i.e.* OCT) is how changes and inconsistencies are dealt with. Static data formats are snapshots of a design at a particular point in the design process, they do not change. Databases are dynamic and thus constantly changing, therefore techniques for dealing with these changes must be developed. This section presents the dynamic behavior of OCT.

There are two types of changes that are automatically propagated by OCT: changes to the bounding box of a master and changes to the format terminals of a contents facet. If the bounding box of the master changes, the bounding box information associated with the instances of that master are updated. If the name of a formal terminal in the contents facet of a view changes, the name is changed in all other facets of the view and all instances of all facets of the view. If a new formal terminal is created in the contents facet, the terminal is added to all other facets of the view and to all instances of all facets of the view. If a formal terminal is deleted from the contents facet, the terminal is deleted from all other facets of the view and all instances of all facets of the view. These operations are performed transparently by OCT. Under certain circumstances, there is one type of change that will not be done transparently. When a formal terminal is deleted, if the equivalent formal terminal in another facet is contained by something other than the facet or contains objects itself, it will not be deleted and instead will be attached to a bag called `*OCT_INCONSISTENT_BAG*`. The `*OCT_INCONSISTENT_BAG*` bag is attached to the facet. Once the attachments to the formal terminal are removed, the next time the facet is opened, the formal terminal will be deleted. This

procedure also applies to actual terminals. If the actual terminal that corresponds to a deleted formal terminal is contained by something other than the instance or contains objects itself, it will not be deleted and instead the actual terminal will be attached to the `*OCT_INCONSISTENT_BAG*` bag. Once the attachments to the actual terminal are removed, the next time the facet is opened, the actual terminal will be deleted.

There are two ways that changes can propagate in OCT: immediate change and deferred change. Immediate changes can occur when the objects that need to be changed are currently in memory. In this case, a simple traversal of the in memory data structures is all that is needed to propagate the changes. If the objects that need to be changed are not in memory, then the changes are performed the next time the objects are brought into memory.

Timestamps are used in OCT to implement deferred changes. When a facet is staged, the timestamps are compared. If the facet is an interface facet, the timestamps are compared with the timestamps on the corresponding contents facet. If the facet contains instances, the timestamps on the instances (actually on non-user-visible master objects) are compared with the corresponding master facets. If the timestamps are different, the facets are checked to see if they need to be updated. There are many timestamps supported by OCT(see the description of `octGetFacetInfo`), but only two types are used for updating: the time of the last change to the bounding box of the facet and the time of the last formal terminal change in the facet.

Besides the inconsistencies that can be encountered when terminals change, inconsistencies can also occur when the masters of instances disappear. Masters can disappear for three reasons: the master has been removed, the master has been moved, or the facet that references the master has been moved. The later case is only a problem if the master is referenced via relative paths; if the master is referenced via an absolute path, the facet can be moved anywhere. When opening up a facet that contains instances, if the masters can not be found, the instances are added to the `*OCT_INCONSISTENT_BAG*` bag[1].

[1]Only one instance for each missing master is added to the bag. Also, the bounding box of the instance information is not stored with the instance, but computed based on the bounding box of the master, so if the master can not be found, the instance has a zero-sized bounding box.

Chapter 10

Monitoring Changes

Change lists allow the monitoring of changes in an OCT facet. A change list consists of zero or more change record objects attached to the change list object. Each change record object records a change in the OCT facet. Change lists are created like all other OCT objects by `octCreate` and change record objects are retrieved by `octGenerate`. A change list object has two fields, one for the types of operations to be monitored and one for the types of objects that operations of the selected type should be recorded. Multiple change lists can be created, each looking for any combination of operation/object pairs. Note that the change lists cannot be specific to an individual object: they work on a class (or on classes) of objects.

The change record contains information about the type of the operation, the xid of the object changed, and the xid of the container of the object changed. `octBB` on the change record will return the bounding box of the object before the operation. In the case of a change record that represents a deleted object, generating the contents of the change record will return the deleted object.

```
/*
 * create a change list to monitor modification (octModify),
 * deletion (octDelete), and creation (octCreate) operations
 * on OCT_INSTANCE and OCT_BOX objects
 */
cl.type = OCT_CHANGE_LIST;
cl.contents.changeList.objectMask = OCT_INSTANCE_MASK|\
                                    OCT_BOX_MASK;
cl.contents.changeList.functionMask = OCT_MODIFY_MASK|\
                                      OCT_DELETE_MASK|\
                                      OCT_CREATE_MASK;
octCreate(&facet, &cl);
```

```
/* do some Oct operations */

/* look at the change list */
octInitGenContents(&cl, OCT_CHANGE_RECORD_MASK, &gen);
while (octGenerate(&gen, &cr) == OCT_OK) {
    switch (cr.contents.changeRecord.changeType) {
        case OCT_CREATE:
            /* get the object that was created */
            octGetByExternalId(&facet,
                    cr.contents.changeRecord.objectExternalId,
                            &object);
            octBB(&object, &bb);  /* bb of created object */
            break;

        case OCT_DELETE:
            /* get the object that was deleted */
            octGenFirstContent(&cr, OCT_ALL_MASK, &object);
           octBB(&object, &bb);       /* bb of deleted object */
            break;

        case OCT_MODIFY:
            /* get the object that was modified */
            octGetByExternalId(&facet,
                    cr.contents.changeRecore.objectExternalId,
                            &object);
            octBB(&object, &bb);  /* bb of current object */
            octBB(&cr, &bb);      /* bb before the modify */
            break;
    }
}

/* remove the change list and change records */
octDelete(&cl);
```

The following are the types of operations in change records.

Change Type	Cause
OCT_CREATE	object is created
OCT_DELETE	object is deleted
OCT_MODIFY	object is modified
OCT_PUT_POINTS	object has points changed
OCT_ATTACH_CONTENT	object is attached (see below)
OCT_ATTACH_CONTAINER	object is attached (see below)
OCT_DETACH_CONTENT	object is detached (see below)
OCT_DETACH_CONTAINER	object is detached (see below)

When an attach or detach operation takes place, a change record is generated based on whether the container and/or contents are in the objectMask for the change list. The masks OCT_ATTACH_CONTENT_MASK and OCT_DETACH_CONTENT_MASK further restrict this by requiring that a change record is generated only if the 'contents' object is in the objectMask for the change list. Similarly, the masks OCT_ATTACH_CONTAINER_MASK and OCT_DETACH_CONTAINER_MASK cause a change record to be generated only if the container object is in the objectMask. If a combination is specified, a change record is generated if either the container or the contents is in objectMask.

The effect is that if you use OCT_{ATTACH,DETACH}_CONTENT_MASK, you get change records where objects in the objectMask are attached or detached from other objects. If you use OCT_{ATTACH,DETACH}_CONTAINER_MASK, you get change records where objects are attached or detached from the objects in the objectMask.

There is not a one-to-one mapping between OCT function calls and change records. Each OCT function call can produce one or more change record objects. For example, the function octCreate can create OCT_CREATE and OCT_ATTACH_CONTENT change records for the object created and an OCT_ATTACH_CONTAINER change record for the object that the new object is attached to. The following lists the OCT functions that create change record objects and the change record objects that are created by each call.

OCT Function	Change Type
octAttach	OCT_ATTACH_CONTENT, OCT_ATTACH_CONTAINER
octDetach	OCT_DETACH_CONTENT, OCT_DETACH_CONTAINER
octCreate	OCT_CREATE, OCT_ATTACH_CONTENT, OCT_ATTACH_CONTAINER
octDelete	OCT_DELETE, OCT_DETACH_CONTENT, OCT_DETACH_CONTAINER
octModify	OCT_MODIFY
octPutPoints	OCT_PUT_POINTS

Deleting a change list also deletes the change records associated with the change list.

There are many uses of change lists, such as an undo facility, and incremental tools (DRC, ERC, extractor, *etc.*). The region searching package for OCT uses

change lists to update the k-d tree data structure when changes occur in the data base. The graphics editor VEM[2] uses change lists for undo, for determining when new layers have been added, and for determining what regions of the screen need to be redrawn after a command.

Problems

The state information associated with a change record corresponds to the state of the database at the time the change record was created. By the time the change record is processed by the application, the state of the database may have changed and the information may be invalid (for example, the `objectExternalId` field in an `OCT_MODIFY` change record will be invalid if the modified object was subsequently deleted). This means that the entire change list must be processed and compressed before programs can use the change information.

The current change list implementation does not persist; this is expected to change in the future.

Chapter 11

Performance

This section presents some basic performance data for OCT on a number of different examples (hand designs and automatically generated designs by multiple module generators).

Disk Space

The following table shows the size (in bytes) for the UNIX file persistent representation of OCT contents facets at various levels of abstraction:

Level	Range
Physical (bytes/object)	20 – 55
Netlist (bytes/object)	25 – 30
Symbolic (bytes/object)	150 – 200
Netlist (bytes/instance)	200 – 270
Symbolic (bytes/instance)	620 – 875

The physical designs only contain geometries, layers, and formal terminals. The netlist designs contain instances, actual and formal terminals, and nets. The symbolic designs contain all that the netlist contains plus layers, geometry, and connectors[1] for implementing the nets[2] [3]. When a facet is staged, its in memory

[1] Connectors are instances that represent vias, and L- and T-junctions.

[2] Note that the current version of OCT shares in memory strings but does not share strings on disk so there is some disk space that can be saved.

[3] See [6] for more information about the representation of physical, netlist, symbolic, and schematic information in OCT.

representation grows to be between 3.5 and 8 times the size of the disk representation of the facet.

Speed

The following shows elapsed time per object for various operations: staging (`octOpenFacet`, reading from disk and allocating memory), unstaging (`octCloseFacet`, flush to disk and deallocating memory), generation (`octGenerate`), and creation (`octCreate`). Two types of objects were used to produce the timings: `OCT_BOX` and `OCT_INSTANCE`. The two types correspond to the simplest and most complex of the object types in OCT. The Objects/Second are specified as a range of numbers with the first (lowest) being for `OCT_INSTANCE` objects and the second (highest) being for `OCT_BOX` objects. The tests were run on a lightly-loaded DECStation 3100 with 24 Megabytes of physical memory running ULTRIX 3.1.

Operation	Objects/Second (Elapsed Time)
Staging	6400 – 7400
Unstaging	5100 – 5200
Generation	59800– 91400
Creation	2800 – 11700

Bibliography

[1] Ellis E. Chang. Effective Clustering and Buffering in an Object-Oriented DBMS. Technical Report UCB/CSD 89/515, UC Berkeley Department of Electrical Engineering and Computer Sciences, Computer Science Division, Berkeley, California, June 1989.

[2] David S. Harrison. VEM: Interactive Graphics for Oct. Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 1989.

[3] David S. Harrison, Peter Moore, Rick L. Spickelmier, and A. Richard Newton. Data Management and Graphics Editing in the Berkeley Design Environment. In *The Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 20–24, Santa Clara, California, November 1986.

[4] Mario Silva, David Gedye, Randy H. Katz, and A. Richard Newton. Protection and Versioning in Oct. In *The Proceedings of the ACM/IEEE Design Automation Conference*, pages 264–269, Las Vegas, Nevada, June 1989.

[5] Rick Spickelmier, editor. *Oct Tools Distribution 3.0*. UC Berkeley Electronics Research Laboratory, Berkeley, California, March 1989.

[6] Rick L. Spickelmier, Jeff L. Burns, and A. Richard Newton. Policy Guides for Oct. Technical Report UCB/ERL M90/??, UC Berkeley Electronics Research Laboratory, Berkeley, California, January 1990.

[7] Rick L. Spickelmier and A. Richard Newton. A Programmer's Guide to RPC. Technical Report UCB/ERL M90/??, UC Berkeley Electronics Research Laboratory, Berkeley, California, January 1990.

[8] Rick L. Spickelmier, Rick Rudell, Peter Moore, David Harrison, Tom Laidig, Wayne A. Christopher, Pete Simanyi, Andrea Casotto, Chuck Kring,

A. Richard Newton, and Alberto Sangiovanni-Vincentelli. Oct Packages. Technical Report UCB/ERL M90/??, UC Berkeley Electronics Research Laboratory, Berkeley, California, January 1990.