Octtools 5.2
Part II: Reference

Electronics Research Laboratory
University of California at Berkeley

May 25, 1993

Contents

Chapter 1

Getting Started

This chapter contains general information about running and installing the Octtools. It assumes some familiarity with UNIX.

1.1 Setting Up the UNIX Environment

To use the Octtools, you will need two things defined in your UNIX environment:

- the `OCTTOOLS` environment variable; and
- the Octtools binary directory in your search path.

Additionally, some tools require general environment variables that some UNIX systems do not initialize, such as `DISPLAY` (for X) and `USER`. On such systems these variables will have to be explicitly defined.

1.1.1 The OCTTOOLS variable

This variable should be set to the machine specific directory in which the tools (binary executables and libraries) reside. This directory can be found directly under the Octtools root directory (after the tools have been installed). Machine platforms, which are determined by both processor architecture and version of UNIX, include:

- `hpux` (Hewlett-Packard HP-UX Version 8 or later and PA-RISC)
- `mips` (DEC Ultrix and MIPS)
- `sun4` (Sun SunOS 4.X and SPARC)

For example, if you are running on a SPARCstation 2, and the Octtools reside in `/usr/cad/octtools`, you should put

```
if ( ! $?OCTTOOLS ) then
    setenv OCTTOOLS /usr/cad/octtools/sun4
endif
```

in your `.cshrc` file.

1.1.2 The Octtools bin directory

In order for UNIX to find the Octtools programs, you will need the Octtools bin directory in your search path. The bin directory is found in the `OCTTOOLS` directory, so

```
set path=($path $OCTTOOLS/bin)
```

in your `.cshrc` file will add the bin directory to your search path.

1.2 Documentation

On-line documentation is available with the **octman** command, a slightly modified version of UNIX **man** that retrieves Octtools documentation. Man pages exist in various degrees of detail and accuracy for most tools. There is also information on Oct library routines (**octman** 3) and formats (**octman** 5). In addition, less formally organized documentation exists on a variety of subjects (**octman** doc).

1.3 Installation

To install the Octtools you need:

- a standard C compiler (from HP, Sun, or DEC – gcc is problematic);
- X Windows, Version X11R4 or later; and
- the Athena Widget Set (`libaw.a`) for X, which is not distributed on all machines. You may have to build it yourself, in which case you can get the source via FTP from gatekeeper.dec.com.

In addition, if you want to build the **vov** tool, you need g++.

Installation is a four stage process:

- getting and setting up the tar files of the release;
- unpacking the tar files;
- setting up the compilation environment for the install; and
- compiling and installing the tools.

1.3.1 Getting the tar files

To get a copy of the Octtools release package contact

```
software@hera.berkeley.edu
```

or

```
Software Distribution Office
Industrial Liaison Program
205 Cory Hall
University of California at Berkeley
Berkeley, CA 94720
(510) 643-6687
(510) 643-6694 (fax)
```

The release package consists of a few text and script files (including `ARCHITECTURE`, `COPYRIGHTS`, `INSTALL`, `PREINSTALL`, and `README`), and a number of compressed tar files that contain source code, technology files, and documentation. All these files should be put in the Octtools root directory. The files `ARCHITECTURE`, `INSTALL`, and `PREINSTALL` are scripts that must be executable (**chmod** 755 ...).

1.3.2 Unpacking the tar files

The `PREINSTALL` script unpacks the tar files. It takes an argument, either `KEEP` or `REMOVE`. If `KEEP` then the compressed tar files are kept; if `REMOVE` then the tar files are removed and just the unpacked files are kept, saving disk space.

The `PREINSTALL` script also verifies checksums on the tar files.

Note that the release package includes items that are not directly part of the Octtools, such as Spice. These items are not unpacked or installed automatically. They also have their own documentation.

1.3.3 Setting up the compilation environment

The `INSTALL` script sets up the platform dependent binary and library directories, and compiles and installs the tools. Two items must be set up correctly for the script to work.

- The `OCTTOOLS` environment variable must be defined. If it is undefined the `INSTALL` script will try to set it, but it is advisable for the variable to be defined in advance. See above for how to set it.
- Two configuration variables in `INSTALL` determine the location of the X11 include files and libraries, `xincloc` and `xlibloc`. As delivered, these are set to values reasonable at UCB, but will almost certainly have to be changed for your system.

In addition, the `ARCHITECTURE` script must run correctly. Run the script and see if it returns the type of machine you are using. If it does not you can either fix the `ARCHITECTURE` script or fix `INSTALL` to compile for your machine type without using `ARCHITECTURE`.

1.3.4 Building the tools

The `INSTALL` script uses a layered sequence of makefiles, with one in the `OCTTOOLS` directory, one in the `src` directory, one in each tool type directory (such as `Layout` or `Framework`), and finally one for each tool. Each of the general (non tool specific) makefiles has an associated `TOOL-LIST` file that is used to select which tools are built. These files can be edited to change the selected tools. The default `TOOL-LIST` files make everything but **BIGOCT**, **BearFP**, **LightLisp**, **OctLisp**, **Unsupported**, **Vov**, **boss**, and **slide-maker**.

Once the environment is configured, simply run `INSTALL` to compile and install the programs, program libraries, technology descriptions, and documentation.

All object files and other effluvia can be removed by connecting to the `OCTTOOLS` directory and typing `make clean`.

1.4 How the Octtools Are Organized

The initial release, before installation, resides in the Octtools root directory under `common`, in the four directories

- `tech` – technology files;
- `lib` – assorted design-related library items;
- `document` – the PostScript form of the manual; and
- `src` – program sources, man pages, test cases, etc.

The installation process creates

- `common/include` (from include files scattered through `common/src`)
- `common/doc` (from doc files scattered through `common/src`)
- `common/man` (from man files scattered through `common/src`)
- `OCTTOOLS/<architecture>` (the platform-specific directory)
- `OCTTOOLS/<architecture>/lib` (which holds lib*.a files)
- `OCTTOOLS/<architecture>/bin` (which holds executables)
- links from `OCTTOOLS/<architecture>` to various directories in `common`

The Octtools universe residing in `common/src` is conceptually divided into two basic classes of things: libraries which, when compiled, end up in `<architecture>/lib`, and tools, which use the libraries and, when compiled, end up in `<architecture>/bin`. Almost all the libraries (called packages), are found in the directories

```
common/src/Packages
common/src/Xpackages
```

and the tools are in

```
common/src/Framework
common/src/Synthesis
common/src/Layout
common/src/Technology
common/src/Simulation
common/src/OtherTools
common/src/Utils
```

Packages: This directory, along with Xpackages and Technology, contains the source code for the libraries. Important packages here are **oct** (the basic Oct database routines), **oh** (additional Oct helper functions and macros), **mm** (the Oct versions of **malloc** and **free**, which don't port very well), and **port** (which contains items related exclusively to porting, primarily `port.h`; note that not all porting problems are solved in `port.h` – individual tools also conditionally compile specialized code). There are several other data structure packages as well, such as **st** (symbol table) and **avl** (avl tree). Most of these packages are documented; either look in `common/doc` or use **octman** (which looks in the `common/doc` directory).

Xpackages: This directory contains X libraries primarily used by **vem**, including **dds** (the dialog box definition system) and the rpc libraries used by vem clients.

Framework: This directory contains general tools and utilities, some of which are oriented toward design, and some of which are Oct database maintenance utilities. Tools include **vem** (the graphical editor), **oct2ps** (a postscript translator), **octman** (man for the Octtools), **bdnet** (the netlist front end), **fabprep** (which prepares an Oct layout for fab), **chipstats** (which compiles statistics on a chip), **octflatten** (which flattens an Oct cell hierarchy), and **attache** (the general Oct facet database editor). These tools range from the almost trivial (**inconsistent**) to the gigantic (**vem**).

Synthesis: This directory contains logic and PLA synthesis tools, including **espresso** and **misII**.

Layout: This directory contains the layout tools, most notably **wolfe/Timberwolf** (the standard cell placer and router) and **mosaico** (the macro cell router), which is a shell script that in turn invokes several other tools.

Technology: The primary item in this directory is **tap**, the technology access package, which is used (by **vem**) to access technology files.

Simulation: The primary items in this directory are **musa**, the multi-level simulator that runs off of the Oct data base, and **steps**, the Oct to Spice translator.

OtherTools: This directory contains miscellaneous items such as **xgraph** and **slide-maker**.

Utils: This directory contains assorted items used for building an Octtools distribution.

1.5 Making a Tool

If you need to build an individual tool, **cd** to the tool's directory, fix `XINCLOC` and `XLIBLOC` in the Makefile to be your X directories (see above), set any necessary platform flags in the CC line, for example

```
CC =/bin/cc -Dhpux
```

and type

```
make <tool-name>
```

or

```
make install
```

to put the executable in `OCTTOOLS/bin`, or

```
make debug-g
```

to get a version with symbols usable by the debugger. If you want a debug-g version, you will also need debug-g versions of the packages (see below).

To build a package (library), **cd** to the package's directory, fix the X directories, and type

```
make install
```

(which will put the package in `OCTTOOLS/lib`), or

```
make debug-g
```

to get a debuggable version (which also puts the package in `OCTTOOLS/lib`).

In general, in a tool or package directory you can type

```
make help
```

for a list of the actions **make** knows about. For more details on Octtools makefiles, if you need to modify or create one, try

```
octman create-octtools-makefile
```

Chapter 2

The General Structure of Oct

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]. It offers a simple interface for storing information about the various aspects of an evolving chip design. A basic unit in a design is the *cell*. A cell is a portion of a chip that the designer wishes to consider as a unit. This can be as small as a transistor or a NAND gate, or as large as the entire floorplan of a CPU. A cell can consist of instances of others cells, such as a NAND gate consisting of several transistors or the floorplan consisting of an ALU, a 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 a routing tool the abstraction might consist of 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 the 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. 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.

The facet is the fundamental unit that you edit in OCT. A particular facet of a view of a cell is opened and edited independently 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 2.1). 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 them.

Being attached is not a particularly strong property: 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. There are a few limitations on what can be attached to what (see the description of `octAttach` for details).

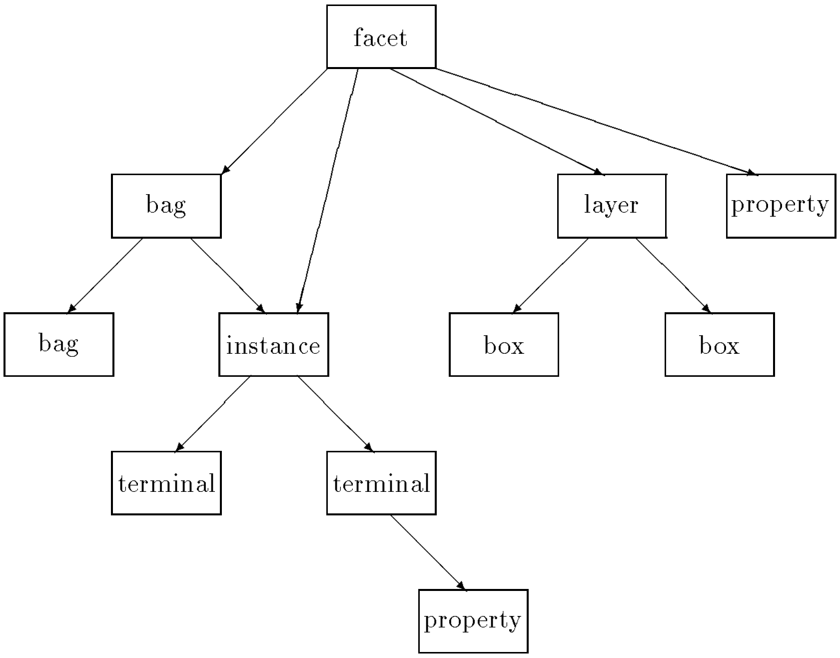


Figure 2.1: Sample Attachments in OCT.

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* describes how terminals and nets are used to represented connectivity. See chapter 3 for detailed information about specific OCT policies.

2.1 The OCT Objects

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

```
struct octObject {
    octObjectType 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;
    } contents;
};
```

`octObject` is used to describe all of the things 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`.

objectIds

Along with the data describing the object and the type of the object, each object contains a field `objectId`. The database uses this ID to find the internal database description of the object. In addition, given the object ID, `octGetById` will return the associated object. This means the object ID can be used as form of pointer to the object, so the entire object need not be kept to reference it, only the id is necessary. These ids are guaranteed to be unique during an OCT session. When an object is deleted, its ID is marked as invalid and may not be used in any further operations; ID's are not reused.

ID's happen to be 32-bit integers, but it is poor programming practice to rely on this fact. ID's should be handled only with the OCT functions `octIdsEqual(), octIdIsNull(), octHashId()`. It is quite likely that in future versions of OCT ID's will be structures with 64 bits or more.

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 facet is to be opened for read, write (and delete old contents, if any), or append, depending on whether `mode` is "r", "w", or "a" respectively. The `version` field is ignored for the moment 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`. Consult the man pages for those functions for further details.

OCT_POLYGON

```
struct octPolygon {
    int  unused;       /* Dummy field, just to avoid having an empty struct. */
};
```

octPolygon describes a polygon. The vertices themselves are accessed the same way as 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. Note that since a polygon is described entirely by its vertices, its definition contains only a dummy field that should never be used. It has been included in Oct to maintain symmetry with the other objects, and because we had doubts about the portability of an empty structure definition.

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

```
#define OCT_JUST_BOTTOM 0
#define OCT_JUST_CENTER 1
#define OCT_JUST_TOP 2

#define OCT_JUST_LEFT 0
#define OCT_JUST_RIGHT 2

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

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 2.2.

OCT_NET

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

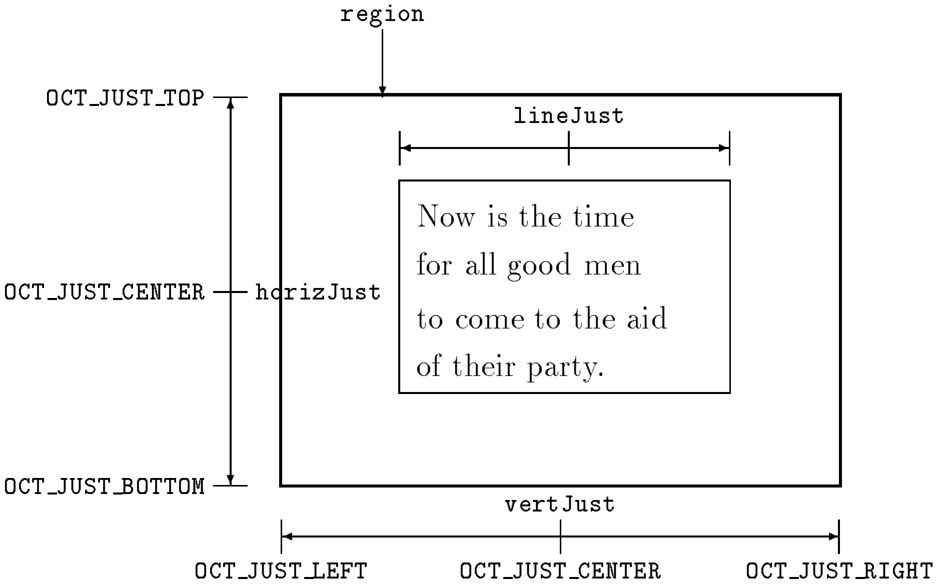


Figure 2.2: The `octLabel` Object

`octNet` describes a net with optional name `name`. The `width` can be used to represent a vector of nets. A net represents a logical connection between the terminals that it contains. In addition it can contain geometry and instances so that the physical implementation can be associated with it. Terminals, geometry, instances and nets are attached and detached using `octAttach` and `octDetach`. They can be accessed using `octInitGenContents` and `octGenerate`.

OCT_TERM

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

`octTerm` describes a terminal of an instance or of a view. If it is a terminal of an instance (an actual terminal), then `instanceId` is the `octId` of the instance to which it belongs. If it 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 created implicitly created when an instance is created, with an actual terminal being created in the instance of a view for each formal terminal in the view itself. Formal terminals can only be created in the `contents` facet of a view, and all other facets may only reference them. The `width` can be used for representing a vector of terminals. To find which nets a terminal is connected to, use `octInitGenContainers` and `octGenerate`. 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. As usual, objects are attached and detached from the bag using octAttach and octDetach, while its contents are retrieved 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. Geometries must be explicitly attached to one (or more if really necessary) layer using octAttach.

In order to retrieve (that is, in Oct lingo, to "generate") all the geometry on one layer a programmer can take a layer object (obtained either by generating all layer objects in the facet or by using octGetByName to get a particular layer) and use octInitGenContents and octGenerate to generate all or part of the geometry attached to that layer. Note that since layers are treated like any other object, it is possible to have more than one layer in a facet with the same name. If you wish to avoid creating a layer with the same name as an already existing layer, you can use the function octGetOrCreate to retrieve a layer with that name if it already exist, or to create it and attach it to the facet if it does not yet exist.

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 {
        long integer;
        double real;
        char *string;
        octId id;
        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, an array of integers, an array of real, or any collection of bytes as defined in octUserValue. Properties can be used to attach any type of information to an object, anything from the last time it was modified to the text of the code that generated it. Because properties can have properties, arbitrarily complex structures can be built.

The value of a property on a object can be obtained by using `octGetByName`, a new property can be added by using `octAttach`, and it can be changed by using `octModify`. Since properties are no different than any other object, it is possible (and sometimes useful) to attach two properties to an object with the same name. To avoid this (if you so desire), you can use the function ''`octCreateOrModify(container, obj)`'', which only creates and attaches `obj` to `container` if it doesn't already exist, otherwise the existing object is modified.

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 facet field is important because it determines the size of the instance bounding box. In most layout the bounding box of the interface facet is identical to the bounding box of the contents facet, but this is not necessarily true for example in schematics, where a large circuit can consists of many transistors and so have a large bounding box in the contents facet, while it may be represented by a small rectangle in the abtraction contained by the interface facet.

The transform consists of a translation and either 1 of the 8 eight manhattan transforms defined in `octTransformType` or a general 2 by 2 tranformation. Instantiating an instance also causes the terminals for that instance to be created as well. See `octTerm` above.

Inside Oct there are routines that check the consistency of the terminals in an instance and the terminals in the contents facet of its master. These routines are activated whenever you open a facet, for all the instances contained in that facet (of course, masters with multiple instances are checked only once). If a facet has many instances of many different types, this checking process may take a long time (a few seconds).

Some tools need to open the master whenever they find an instance. For example, **vem** needs to do that in order to find out how to display the instance. In general, the interface facet is much smaller than the contents facet, and therefore it is much faster to open. If you want to increase the efficiency of your tools, you need to have some understanding of the mechanism used by OCT to maintain terminal consistency (another example to study is offered by the tool **tapPP**: `octman tapPP`).

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 2.10 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 2.10 for more information.

2.2 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_CHANGE_LIST_MASK	OCT_CHANGE_RECORD_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`.

octInitGenContentsSpecial

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

The *special* generators take a snapshot of the attachments of objects of type specified by `mask` to `obj` If you attach or detach objects in the middle of the generation sequence, the generator will not see these changes. *Special* generators are especially useful in RPC applications, because they are up to 10 times faster than regular generators.

Note/bug: in regular OCT (not RPC), a special generator is exactly the same as a regular generator. This function is provided so that code written for RPC applications works with the `liboct.a`.

octInitGenContainersSpecial

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

See comment for `octInitGenContentsSpecial()`.

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());
```

2.3 Useful packages

In contrast with the simplicity of the **oct** procedural interface, where each function such as `octCreate()` operates on any type of object, the package **oh** provides a large number of very specialized functions, too many to list here. For example, in order to create objects, **oh** offers the following functions (actually implemented as macros):

```
ohCreateBox();
ohCreateInstance();
ohCreateNet();
ohCreateTerm();
ohCreateBag();
ohCreatePropStr();
ohCreatePropInt();
ohCreatePropReal();
```

In most OCT applications, these procedures allow a more compact and more readable code. Procedures from **oct** and **oh** can be freely intemixed. Please refer to `$OCTTOOLS/doc/oh.doc` for more documentation on **oh**.

Other packages that you will find useful, even if you are not dealing with **oct** are **utility**, **mm**, **port**. Please refer to their respective document in `$OCTTOOLS/doc`.

2.4 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. 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_INCONSISTENT: A persistent version of the facet exists and has been read in, however some of the masters have not been found or terminals 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[1]. 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[2] 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 {
    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):

[1] 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.

[2] 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.

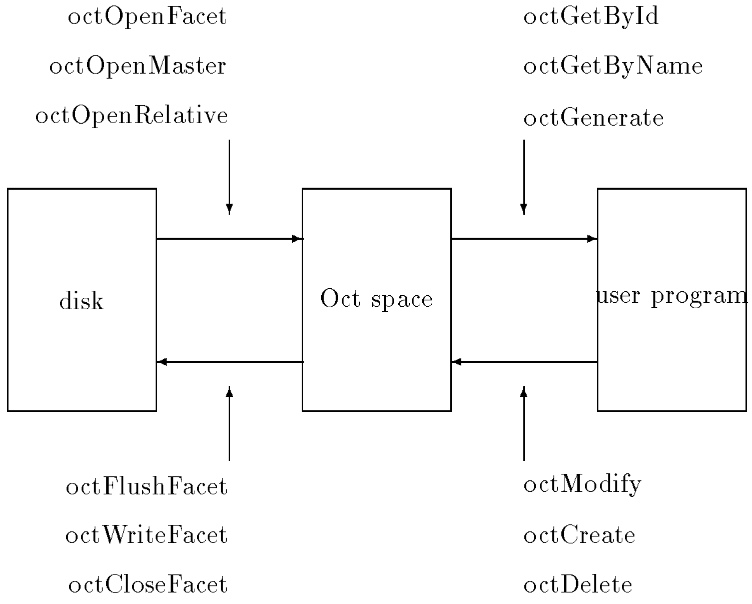


Figure 2.3: OCT Operations and the Environment

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

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

2.5 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 */
```

2.6 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.

2.7 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.

2.8 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).

2.9 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[3].

2.10 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 */
```

[3] 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.

```
            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_M` 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 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.

2.10.1 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.

2.11 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[4] for implementing the nets[5] When a facet is staged, its in memory 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.

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

[5] 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.

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

Chapter 3

Octtools policies

OCTprovides a *mechanism* for representing CAD data but places no meaning on the data. *Policy* is used for assigning meaning. For example, OCT has objects that represent layers and geometry, but it does not specify how to relate them 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 it does not specify how connectivity is represented. It is the *policy* that describes how terminals and nets are used to represented connectivity.

This document describes the policies that are currently in the tools in the **Octtools 5.2** distribution.

3.1 Oct Generic Policy

Oct supports arbitrary attachments among objects. This document specifies the legal attachments among objects that are generic across *physical*, *symbolic*, and *schematic* representations.

Each view of a cell has two facets, a **contents** facet and an **interface** facet. The **contents** facet contains the *definition* of the view. The definition contains the objects that comprise the view in detail (actual geometries and hierarchy). The **interface** facet contains an *abstraction* of the view. The abstraction is intended to contain the minimal information needed to describe the cell to higher-level cells and tools (*e.g.,* terminals, routing regions, annotation).

The abstraction is normally derivable from the definition, using tools such as **vulcan** and **symgen**. However, there exist cases in which the interface facet has information that does not exist in the contents. This is information that is needed only by tools that look at the view "from the outside."

3.1.1 Contents Facet

The contents facet contains the detailed description of the view.

Formal Terminals

Formal terminals define the locations where connections can be made to the cell. Formal terminals have a name and optionally a location and size. Formal terminal definitions are stored in Oct via *octTerm* objects. Terminal objects in Oct are logical objects in the sense that no geometric information is present in an octTerm object. Rather, the geometric information that describes a terminal is attached to the terminal, and is referred to as the terminal's implementation. For clarity, this distinction is emphasized in this document by referring to an octTerm as a terminal *definition*, and the associated geometry as the terminal's *implementation*. The formal terminal definition contains the name of the terminal.

All formal terminal names must be unique (but Oct does not enforce this condition). Some tools may break if the names contain strange characters such as `" ~ ; : #` . The formal terminal definitions also appear in the interface facet and are included automatically by Oct upon creation of the interface. Due to this inheritance mechanism, the contents facet must be created before the interface facet.

Layers and Geometry

The layer of a geometry is specified by attaching the geometry to an *octLayer* object. The octLayer object is attached to the facet. No geometry can be directly attached to the contents facet. The names for each octLayer are constrained by the technology description (see Section 3.1.1). Labels are attached to a layer to give them a display color.

Properties and Bags

All Oct facets are required to have three *octProp* objects attached to them: `TECHNOLOGY`, `VIEWTYPE`, and `EDITSTYLE`.

`TECHNOLOGY:` the name of the technology used by the facet, it is used by the technology access package **tap** to determine technology-specific information. The names of the allowed technologies are usually the names of directories in `$OCTTOOLS/tech` (the value `<self>` denotes a **tap** technology description).

`VIEWTYPE:` along with `TECHNOLOGY`, is used by **tap** to determine the names of the layers in the technology[1]. `VIEWTYPE` normally has one of the following values: `PHYSICAL`, `SYMBOLIC`, or `SCHEMATIC`.

`EDITSTYLE:` used by the graphics editor **vem** to determine what editing commands will be allowed for the facet. The allowed values for `EDITSTYLE` are `PHYSICAL`, `SYMBOLIC`, and `SCHEMATIC`. These values correspond to the available editing styles in **vem**.

3.1.2 Interface Facet

The interface facet is the abstraction of the contents facet. An interface facet contains the following explicitly attached objects:

- Formal terminal definitions
- Protection frames
- Layers
- Properties and bags

Table 3.1 summarizes the interface facet. The attachments are shown in Figure 3.1. Note that there are no instances or nets in the interface facet. The program **vulcan** generates interfaces from contents[2].

[1] **tap** stores information on the valid layers, how to draw them on various devices, the minimum width for each layer, and the design-rules

[2] Currently, **vulcan** does not generate interface facets for *schematic* views (use **symgen** instead) and it does not do all that might be necessary to make a complete interface, *i.e.,* does not put `TERMTYPE`, `DIRECTION`, and `GLOBALNET` properties on the formal terminal definitions in the interface.

Object	Containers	Contains
interface facet		formal terminals protection frames layers properties and bags
formal terminal	facet	terminal frames properties and bags
protection frame	facet layer	
layer	facet	protection frames terminal frames
properties and bags	facet formal terminal	
terminal frame	formal terminal layer	

Table 3.1: Interface Facet

Object	Containers	Contains
interface facet		JOINEDTERMS bag formal terminals
JOINEDTERMS bag	interface facet	JOINED bags
JOINED bags	JOINEDTERMS bag	formal terminals

Table 3.2: Joined Terminal Attachments

Formal Terminals

The formal terminal definitions are inherited from the contents facet when the interface facet is created. Oct does not propagate any other information from the contents to the interface.

The implementations of the formal terminals in the interfacet facet, or *terminal frames*, are geometries that are attached to the appropriate formal terminal definitions and layers. Terminal frames are restricted to *octBox* objects. Multiple octBox objects can be used to implement a single formal terminal, however the center of each *octBox* must be inside the largest octBox (*i.e.,* they cannot be disjoint). These differ from the implementations of the formal terminals in the contents facet, which can be any geometric shape or an actual terminal (the exact restrictions are defined by the policy for that view).

Formal terminals that are electrically equivalent (*e.g.,* connected to the same net inside the cell) are contained in a `JOINED` bag. Each `JOINED` bag is contained by the `JOINEDTERMS` bag that is attached to the facet (see Table 3.2). The interface facet generator **vulcan** will do this automatically[3].

Protection Frames

Protection frames are used to abstract the body of the cell. Each frame is attached to the interface facet and to the appropriate layer. The protection frames for a layer are composed of zero or more *octPolygon* objects that are restricted to *manhattan* shapes. Note that protection frames can be differentiated from terminal frames by generating only the geometries that are directly attached to the facet. The area taken up by terminal frames is **not** included in the protection frames.

There is an extra layer in the interface named `PLACE` that contains a single manhattan octPolygon object. This object is the union of all protection frames, grown and shrunk by enough to form a single polygon. See Figure 3.2 for an example contents facet and it's corresponding protection and terminal frames.

[3] **vulcan** will not do this for *physical* views since there is no explicit interconnectivity information available.

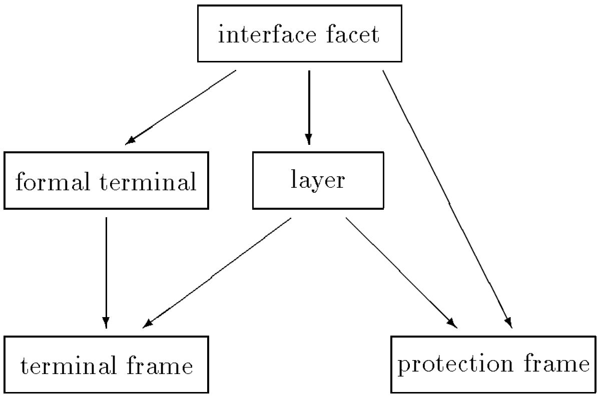


Figure 3.1: Interface Attachments

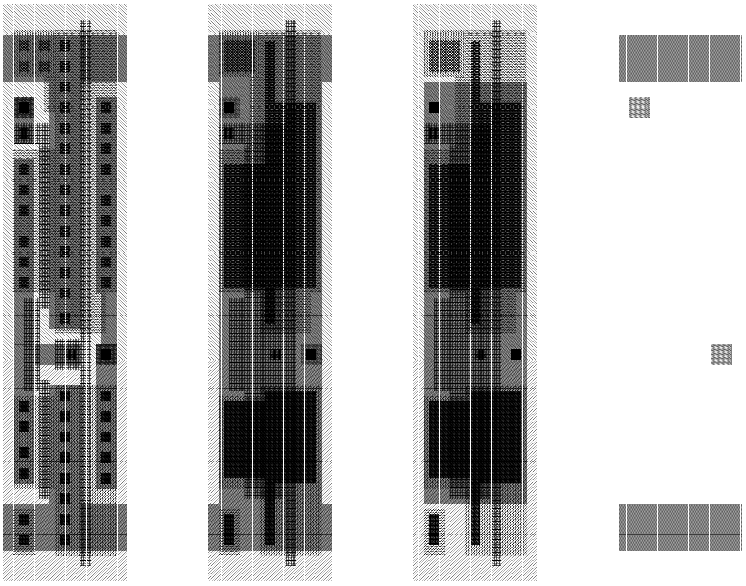


Figure 3.2: Contents, Interface, and Corresponding Protection and Terminal Frames

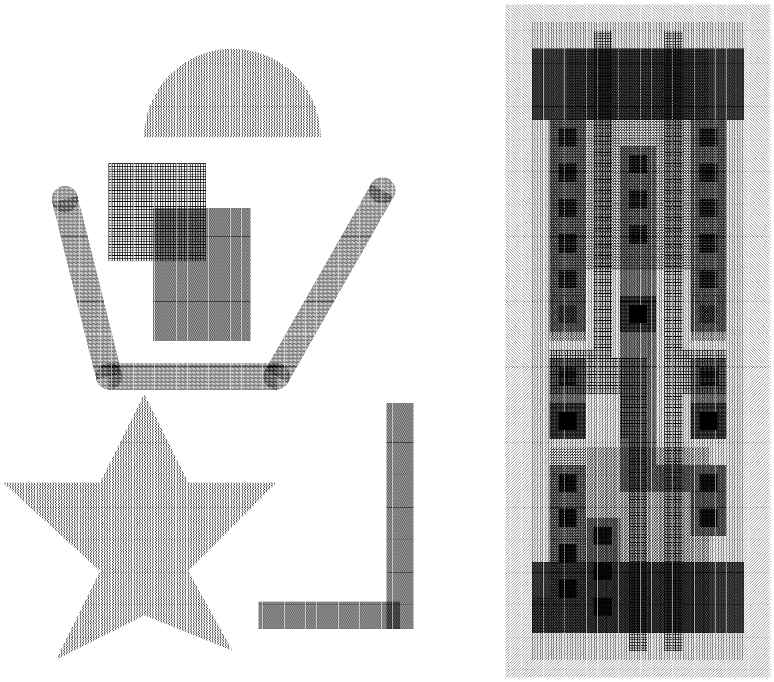


Figure 3.3: Example Physical Designs

Layers

Layer objects are represented by an *octLayer* object and are attached explicitly to the interface facet. Every geometry, *i.e.,* every protection frame object and every terminal frame object, is attached to exactly one layer object.

Properties and Bags

In addition to the `TECHNOLOGY`, `VIEWTYPE`, and `EDITSTYLE` properties, there are a number of properties and bags that can be in an interface facet. For example:

- Attached to the facet:

  `CELLCLASS:` class of the cell, which is either `LEAF`, `MODULE`.

  `CELLTYPE:` type of the cell, *e.g.,* `MOSFET`, `COMBINATIONAL`.

- Attached to the formal terminals:

  `TERMTYPE:` type of the terminal, *e.g.,* `TRISTATE`, `SIGNAL`.

  `DIRECTION:` signal direction of the terminal, *e.g.,* `INPUT`, `OUTPUT`.

See the individual policy descriptions for a description of the properties and bags that have specific meaning for that policy.

3.2 Oct Physical Policy

This section specifies the legal objects and the legal attachments among objects for *physical* views. It is stated as a set of additions/restrictions to the Oct *generic* policy. Figure 3.3 shows example physical designs.

No interconnectivity can be specified in physical views and thus the *octNet* object is not allowed in physical views.

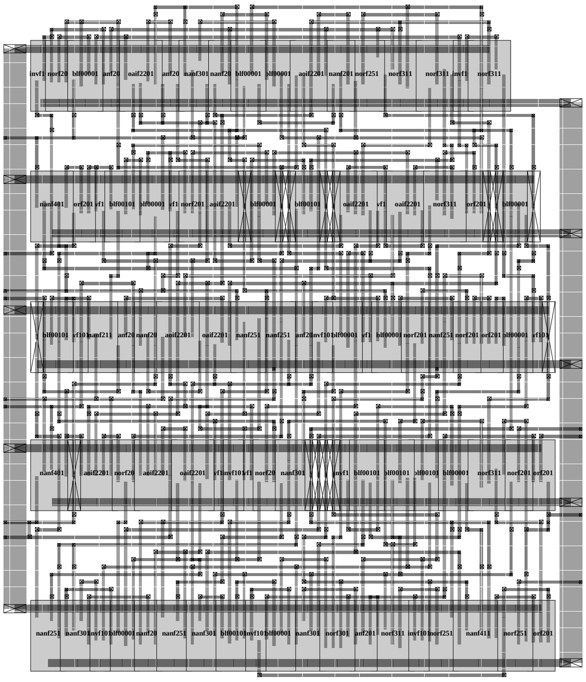


Figure 3.4: Example Symbolic Designs

In both the contents and the interface facet, the value of the `EDITSTYLE` and the `VIEWTYPE` properties must be `PHYSICAL`.

3.3 Oct Symbolic Policy

This section specifies the legal objects and the legal attachments among objects for *symbolic views* as additions/restrictions to the Oct *generic* policy. Figure 3.4 shows example symbolic designs.

3.3.1 Contents Facet

The contents facet is fundamental and must exist before the interface facet. The contents facet contains objects that are directly attached to it and objects that are not. The following are the directly attached objects:

- Formal terminal definitions
- Instance class bags
- Instances
- Nets
- Layers
- Properties and bags

These objects are the **only** objects that are directly, or **explicitly**, attached to contents facets. Other objects that are part of the facet are attached to the objects listed above rather than to the facet. Table 3.3 summarizes the objects and attachments of a contents facet. The attachments of a simple contents facet are shown in Figure 3.5.

Object	Containers	Contains
contents facet		formal terminals cell instances connector instances INSTANCES bag CONNECTORS bag nets layers properties and bags
formal terminal	facet net	actual terminal properties and bags
INSTANCES bag	facet	cell instances
CONNECTORS bag	facet	connector instances
cell instance	facet INSTANCES bag	instance terminals properties and bags
connector instance	facet CONNECTORS bag	connector terminal
net	facet	segments instance terminals formal terminals properties and bags
layer	facet	segments
actual terminal (instance terminal)	cell instance formal terminal nets segments	properties and bags
actual terminal (connector terminal)	connector instance formal terminal segments	
segment	net layer	actual terminals
properties and bags	facet formal terminals cell instances nets instance terminals	

Table 3.3: Contents Facet

Formal Terminals

In symbolic contents facets, formal terminals are **always** implemented by attaching *actual terminals*[4] to them. The formal terminal definitions also appear in the interface facet. They are included in the interface automatically by OCT upon creation of the interface. Due to this inheritance mechanism, the contents facet must be created before the interface facet.

Connectivity is represented (in part) by associating terminal **definitions** with nets. Formal terminals are directly attached to nets. Using OCT terminology, a formal terminal definition is **contained** by the facet (and possibly a net) and the formal terminal **contains** an actual terminal.

Instance Class Bags

There are two classes of instances in symbolic views: **cell instances** and **connector instances**. As explained below, connector instances are used in implementing the interconnections between major instances. Two bags are used in contents facets to differentiate the two types of instances. All major instances are attached to the bag named `INSTANCES` , and all connector instances are attached to the bag named `CONNECTORS`. These two bags are attached to the contents facet.

Instances

All instance objects in symbolic views appear in contents facets; no instances are allowed in interface facets. If information from the master of an instance is needed, the interface facet of the master should be consulted. For symbolic views, instance masters may be views of type symbolic or views of type physical. Physical views of cells are at the leaf-level of the symbolic view instance hierarchy. However, such physical views may contain instances of other physical views.

As mentioned in the previous section, the two classes of instances in symbolic views are the cell instances and the connector instances. These are usually referred to as *instances* and *connector instances*, respectively. The class of an instance is distinguished via the bag it is attached to. In addition, all instances are attached to the facet. Connector instances are used in net implementations and are described in the following subsection. All other instances, such as transistors, macro cells, and standard cells, are cell instances. In order to determine if a new instance belongs in the `INSTANCES` or `CONNECTORS` bag, the value of the `CELLTYPE` property in the interface facet of the master is checked; if the value is `CONTACT` it is a connector instance.

Both cell and connector instances have terminal definitions attached to them, and they are called *actual terminal* definitions. The actual terminal definitions of an instance correspond to the formal terminal definitions of the instance's master, and are inherited from the master. The association is by name and the name is currently the only way to go from an actual terminal to the corresponding formal terminal. OCT creates the actual terminal definitions for each instance when the instance is created. Connectivity with respect to actual terminals is expressed by attaching them to nets.

The implementation of an actual terminal is **not** inherited from the master; only the actual terminal definitions are inherited. Accordingly, terminal implementations are never attached to nets. To extract the information that describes an implementation of an actual terminal, the user must open the interface facet of the instance's master cell.

Nets

Nets in symbolic views are represented by an *octNet* object with segments, formal terminal definitions, and actual terminal definitions attached. All routing is expressed in terms of single segments which are stored as two-point *octPath* objects. The **only** geometric objects allowed in the contents are these segments. When multiple segments intersect, for example at *tee junctions* or *ell corners*, a **connector instance** must be placed at the intersection. The master of a connector instance consists of a formal terminal definition and an implementation geometry(s) on the same mask layer(s) as the segments. For example, a connector instance placed at the junction of two segments on the metal layer has a single terminal implementation geometry on the metal layer. A connector instance at

[4] Actual terminals are described in Section 3.3.1

the junction of a metal segment and a poly segment has terminal implementation geometries on the metal, poly, and cut layers. The terminal of a connector instance is referred to as a *connector terminal*; for clarity, all actual terminals other than connector terminals, *i.e.,* those from cell instances, are called *instance terminals*. The connector terminals are like all other actual terminals in that they are inherited from the interface facet of the appropriate master.

Three types of objects are **directly contained** by nets: formal terminal definitions, actual terminal definitions of **cell** instances, and segments. Actual terminals of **connector** instances **are not** contained by nets. However, if a connector terminal is promoted by a formal terminal, the formal terminal **is** attached to the net. Each segment **contains** the two actual terminals associated with it; that is, the two actual terminals whose implementation geometries intersect the two ends of the segment. The net attachments are summarized in Figure 3.6.

The configuration of the net attachments allow each application program to extract the portion of the net information of interest to it. The application that needs the detailed net implementation information (*e.g.,* for net-length estimation) can follow the *(net)→(segment)→ (actual terminal definition)* attachments. If a netlist is to be determined, the *(net)→(actual terminal definition) →(instance)* attachments may be used.

Layers and Geometry

The only geometry allowed in the contents of a symbolic view is the geometry that belongs to net implementations, *i.e.,* the segments described above. Accordingly each *octPath* object is attached to its mask layer object.

Properties and Bags

The `EDITSTYLE` property must have the value `SYMBOLIC` for symbolic views. For a complete list of allowed properties and bags, see the the Symbolic Policy Manual Pages. The values of certain properties and bags in a master can be overridden by placing a property or bag of the same name on the instance. The Symbolic Policy Manual Pages specify which parameters can be overridden.

3.3.2 Interface Facet

Properties and Bags

The `EDITSTYLE` property must have the value `SYMBOLIC` for symbolic views.

3.4 Oct Schematic Policy

This section specifies the legal objects and the legal attachments among objects for *schematic* views. Schematic views are a modification of *symbolic* views and thus will be presented as a set of additions and restrictions to the Oct symbolic policy. Figure 3.7 shows example schematic designs.

3.4.1 Contents Facet

Layers and Geometry

Only one layer is allowed in the contents facet of a *schematic* view[5].

This layer is for interconnection, and is named `WIRING`.

[5] This is not strictly true since at the lowest level: the contents facet will be a symbol and set of terminals composed of geometry on the `SYMBOL` and `WIRING` layers.

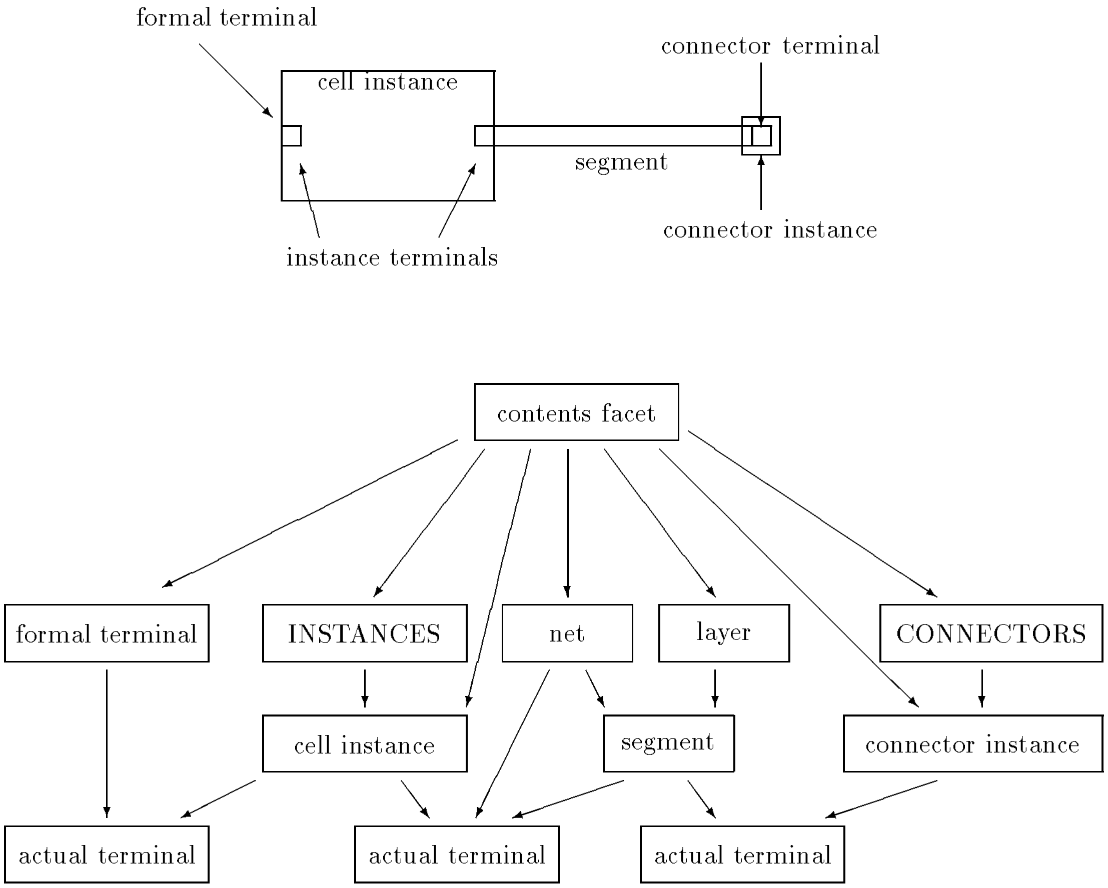
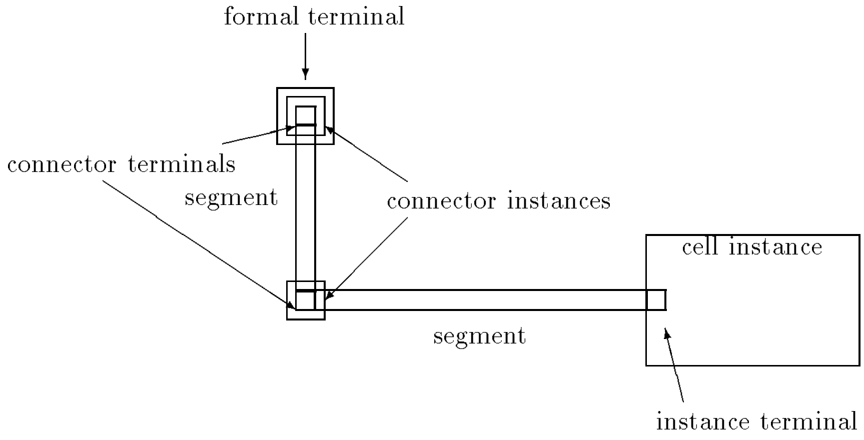


Figure 3.5: Attachments for a Simple Symbolic Contents Facet



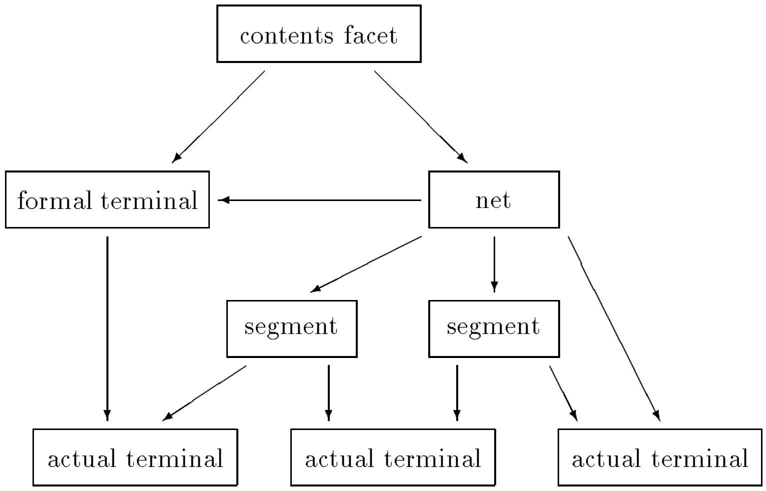


Figure 3.6: Net Attachments

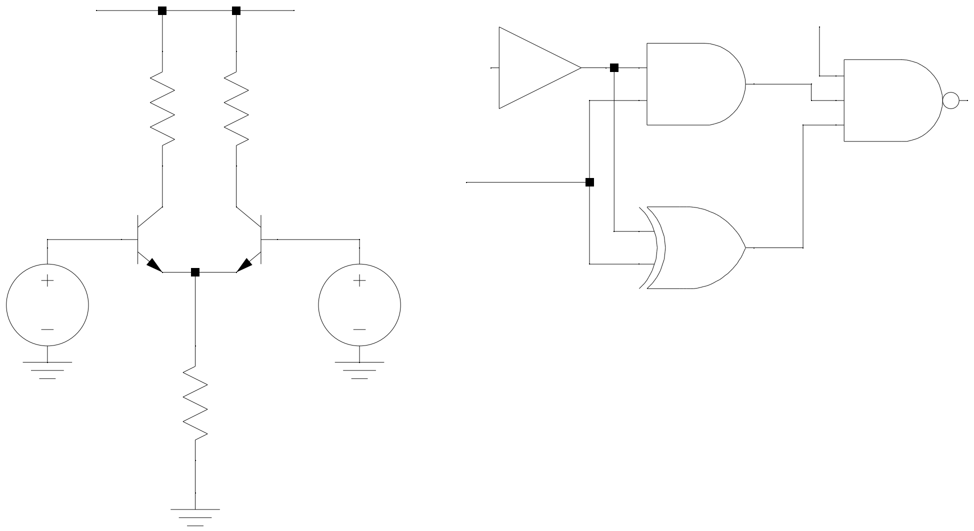


Figure 3.7: Example Schematic Designs

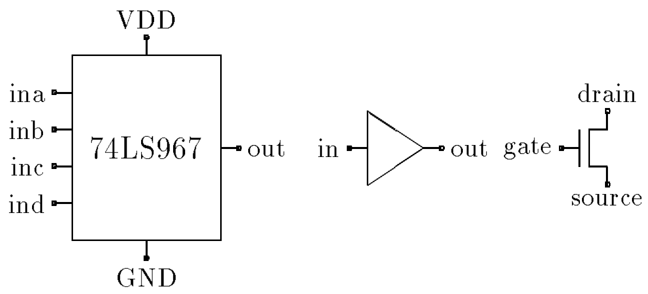


Figure 3.8: Example Interface Symbols

Interconnect

Interconnectivity is specified in exactly the same way as in symbolic views, with the exception of the implementation. Since there is only one layer for interconnection, there are no *vias*, only connector objects for specifying *t-junctions* and *ells* (bends). Like in *symbolic* views, all interconnect is restricted to single-segment paths, however in *schematic* views there is one further restriction that all paths be *zero-width*.

Busses and Bundles

Busses and bundles represent collections of nets. We have a policy on this, expect more documentation in the future.

Properties and Bags

The value of the `EDITSTYLE` property must be `SCHEMATIC`.

Displayable Properties

When viewing a schematic, either in a graphics editor, such as **vem**, or as a plot, there are various annotations on an instance or master that should be displayed. We have a policy on this, expect more documentation in the future.

3.4.2 Interface Facet

Layers and Geometries

Two layers, named `SYMBOL` and `WIRING`, can exist in interface facets of *schematic* views. The `SYMBOL` layer is used for building symbols. Any geometric shape can be attached to the `SYMBOL` layer. Terminal frames must be *octBox* objects attached to the layer named `WIRING`. The `WIRING` layer can only be used for terminal frames. In general, the interface facet will be an abstract *symbol* that represents the contents facet, see Figure 3.8 for examples.

Protection Frames

There are no protection frames in *schematic* views. See Section 3.4.2 for details on the geometries allowed in the interface facet.

Off-Page Connections

There are two mechanisms for supporting off-page connections. They are specified by the `LOCALNET` and `GLOBALNET` properties.

`LOCALNET`: when an instance is placed, for each interface formal terminal definition that has a `LOCALNET` property attached, the actual terminal will be connected to the `octNet` object with a name that is the value of the `LOCALNET` property. If an *octNet* object with that name does not exist, it will be created and the actual terminal will be attached. If the terminal is placed such that it should be merged into an already existing net, the net will be renamed to the value of the `LOCALNET` property. If the `LOCALNET` property has no value, the editor will query for a value.

`GLOBALNET`: Same as `LOCALNET` with the addition that the *octNet* object will be attached to a formal terminal definition of the same name and a `GLOBALNET` property, with the same value, will be attached to the formal terminal definition. If the formal terminal definition does not exist, it will be created. Now with the `GLOBALNET` property on a formal terminal of the cell, when the cell is instatiated at the next higher-level, this process will be repeated (thus propagating the connectivity information all the way up the hierarchy). `GLOBALNET` is appropriate for supplies and ground.

Properties and Bags

The value of the `EDITSTYLE` property must be `SCHEMATIC`.

Displayable Properties

When viewing a schematic, either in a graphics editor, such as **vem**, or as a plot, there are various annotations on an instance or master that should be displayed. This function is performed by the package **lel** and all the details are explained in `$OCTTOOLS/doc/lel.doc`.

3.5 Policy on Oct and Spice

This document specifies the additions to the *symbolic* and *schematic* polices for representing **spice** models and parameters.

This policy is used by **steps** and **rpc-steps**, which create **spice** files from a symbolic layout or from a schematic diagram. The policy has not been finalized and may change in the future. We are interested in any comments you might have concerning representing simulator input data in Oct.

3.5.1 Model Bags

In the interface facet of each primitive, there can be a bag containing modeling information. This bag has a name formed by concatenating the value of the `CELLTYPE` property with `MODEL` (*e.g.,* `BJTMODEL`, `RESISTORMODEL`, *etc.*). Since different simulators seem to have irreconcilable differences in the paramters their models take, we have a separate bag in the model bag for each simulator. This bag is named for the simulator.

Inside this bag are properties, whose names should make sense to the simulator. So far, we've only dealt with scalar floating-point values, but arrays of floats and strings should be OK, if that's what the simulator wants. A simple example of the properties on a `BJT` is given in the man page for **octprop** (although it talks about the **SPICE** bag, which should be the **SPICE3** bag). If any parameter is missing, or if no model bag exists or no simulator-specific model bag exists, the simulator should use its own defaults (this wholesale defaulting may not be right for devices like FET's and BJT's, but is good for resistors and capacitors).

3.5.2 Properties

Note that all properties described below are attached to the interface of the master. All but `CELLTYPE` and `*TYPE` can be overriden on the instance.

BJT ..

Name	Type
BJTAREA	real valued property (in square meters)
BJTTYPE	string valued property (NPN, PNP)
BJTMODEL	bag

CAPACITOR ..

Name	Type
CAPACITANCE	real valued property (in farads)
CAPACITORMODEL	bag

DIODE ..

Name	Type
DIODEAREA	real valued property (in square meters)
DIODEMODEL	bag

INDUCTOR ..

Name	Type
INDUCTANCE	real valued property (in henries)
INDUCTORMODEL	bag

MOSFET ..

Name	Type
MOSFETTYPE	string valued property (NENH, PENH, NDEP, PDEP)
MOSFETWIDTH	real valued property (in meters)
MOSFETLENGTH	real valued property (in meters)
MOSFETDRAINAREA	real valued property (in square meters)
MOSFETSOURCEAREA	real valued property (in square meters)
MOSFETDRAINPERIM	real valued property (in meters)
MOSFETSOURCEPERIM	real valued property (in meters)

RESISTOR ..

Name	Type
RESISTANCE	real valued property (in ohms)
RESISTORMODEL	bag

SOURCE (for independent sources) ..

Name	Type
SOURCETYPE	string valued property
SOURCEDCVALUE	real valued property (in Volts of Amperes)
SOURCEMAGNITUDE	real valued property (in Volts of Amperes)
SOURCEPHASE	real valued property (in degrees)
SOURCETRANSIENTMODEL	bag

CONTROLLEDSOURCE (for linear controlled sources) ..

Name	Type
CONTROLLEDSOURCETYPE	string valued property (VCCS, VCVS, CCCS, CCVS)
CONTROLLEDSOURCEGAIN	real valued property

3.6 Symbolic/Schematic Properties

This section describes how MOSFET's and logic cells are represented in *symbolic* and *schematic* views. The names and values of these properties have evolved over the years in a somewhat disorganized fashion, which accounts for some inconsistency in conventions: some names separate words with a dash "-" others with an underscore "_" while others do not separate the words. A clean-up is in order; may be in the next release ...

3.6.1 Mosfets

MOSFET's have the following properties attached to the *interface* facet:

Name	Value
EDITSTYLE	PHYSICAL
VIEWTYPE	PHYSICAL
CELLCLASS	LEAF
CELLTYPE	MOSFET
MOSFETTYPE	NENH, PENH, NDEP, or PDEP
MOSFETWIDTH	real number (in meters)
MOSFETLENGTH	real number (in meters)

MOSFETS have the following terminal names: `DRAIN`, `SOURCE`, `GATE`, `GATE1`. `GATE` and `GATE1` are in a `JOINED` bag.

3.6.2 Logic Cells

Logic cells have the following properties attached to the *interface* facet:

Name	Value
CELLCLASS	LEAF
CELLTYPE	COMBINATIONAL or MEMORY

Each formal terminal in the interface facet has the following properties:

Name	Value
TERMTYPE	SIGNAL, SUPPLY, GROUND, ...
DIRECTION	INPUT, or OUTPUT

The `DIRECTION` property is only needed for terminals with `TERMTYPE` equal to `SIGNAL`.

COMBINATIONAL

For each formal terminal in the interface facet with a `DIRECTION` property with the value `OUTPUT`, a `LOGICFUNCTION` property must be attached. The `LOGICFUNCTION` is defined as follows (the `LOGICFUNCTION` is overrideable on the actual terminal):

function	(operator argument+)
operator	`+, *, ^, !`
argument	function
	terminal name

Examples of `LOGICFUNCTION` values are: `(+ in1 in2)`, `(! in1)`, and `(! (+ (* a b) (* c d)))`.

MEMORY

Memory cells are latches, flip-flops, and, believe it or not, transmission gates too (the reason being that transmission gates have a control signal, an input and an output, like a latch).

Name	Value
CELLTYPE	MEMORY
SYNCHMODEL	TRANSPARENT-LATCH, MASTER-SLAVE-LATCH, or TRANSMISSION-GATE

For each formal terminal in the interface facet with a `TERMTYPE` property with the value `SIGNAL`:

Name	Value
SYNCHTERM	OUTPUT, OUTPUTBAR, INPUT, INPUTBAR, CONTROL, CONTROLBAR, SET, RESET
or ASYNCHTERM	SET, RESET
ACTIVE_LEVEL	HIGH (default) or LOW

Notice that this set of properties allows inconsistent descriptions (for example, it it not clear what the meaning of terminal with SYNCHTERM=CONTROLBAR and ACTIVE_LEVEL=HIGH would be). Look at the latches in the *MSU* standard cell library for examples.

RAM

In the interface facet of a RAM cell add the following properties:

Name	Value
CELLTYPE	MEMORY
SYNCHMODEL	not used
MEMORYMODEL	RAM

The terminals should be annotated with the following properties:

Name	Value
RAM_TERM	READ_ENABLE, WRITE_ENABLE, ADDRESS, DATA
ACTIVE_LEVEL	HIGH (default) or LOW

LED

The simulator **musa** can represent LED's graphically, a feature that can add a great deal of insight in the operation of a circuit. In order for **musa** to understand an LED, it must have the following properties:

Name	Value
CELLTYPE	I/O
I/O-MODEL	LED
LED_COLOR	RED, GREEN, or YELLOW

An LED must have two terminals, called ANODE and CATHODE.

Chapter 4

Guidelines for Inclusion of Tools and Packages

In order to ease the building and testing of the next release, please follow the guidelines listed below.

1. Do not use the fact that (currently) cells are directories, views are directories, and facets are files. In other words, do not check for the existence of directories and files, and do not create files in the cell or view directories.

2. Functions, variables and macros exported by packages must begin with a prefix that represents the package. For example, all functions and macros in the **region** package begin with `reg`, and all functions and macros in the **oh** package begin with `oh`. If possible, do not use "_" to separate words in function and variable names, use capitalization (`octAttach` instead of `oct_attach`.

3. Each package should have one and only one exported header file. The information in the header file should be surrounded by the following:

   ```
   #ifndef <package>_H
   #define <package>_H

   /* package information */

   #endif
   ```

   Where `<package>` is the name of the package in capital letters.

4. There will be a number of new **tap** features that will improve access to cell libraries and get rid of the need for text files in the technology directories. Use `tapGetDirectory` to find the technology directory, do not hardwire `$OCTTOOLS/tech` into your code. Do not hardwire a default technology into your tool/package. Let **tap** determine the technology. If **tap** can not determine the technology or technology directory, complain and exit. When using **tap**, you should also use the **option** package so that you get the **-=T** flag that allows the default location of the technology directory to be overriden.

5. Use the packages as much as possible; do not reinvent the wheel. The three most useful packages seem to be: **port**, **utility**, and **oh**. The **port** package hides function name, return value, and include file location changes for the various versions of UNIX. The **utility** package provides a number of useful macros (for memory allocation) and functions (tilde expand, path search, *etc.*). The **oh** package is a set of Oct helper routines that simplify common operations. Look in `$OCTTOOLS/doc` for the documentation on existing packages.

6. You should use the **mm** package for memory allocation (unless you are using C++ or g++). This is plug compatible with the standard UNIX memory allocation routines (`malloc`, `realloc`, and `free`), but provides a greater degree of safety (detects `malloc` returning 0, corrupt free list, freeing something that has already been freed, freeing something that has never been allocated, *etc.*). All you need to do to use this is to put `${OCTTOOLS}/lib/libmm.a` at the end of your load line (this will be done automatically by using **create-octtools-makefile**).

7. Do not overuse `OH_ASSERT` or whatever your favorite OCT macro is called. For example, if you are looking for the `CELLCLASS` property and it is not found, print out an informative error message, such as:

   > Can not find the CELLCLASS property on the interface facet of cell:view. The facet does not conform to symbolic policy.

   Do not print out:

   ```
   assertion failure: readoct.c, line 35: octGetByName failed
   ```

   Specifically, do not use `OH_ASSERT` on `octGetByName` or `octOpenFacet`.

8. The tools and packages must lint, compile, and run under ULTRIX, SUN, and BSD. Use of the **port** package will significantly improve your chances. Access to BSD and SUN machines will be arranged.

   The following items will improve portability across machines:

   (a) Use `strchr (strrchr)` rather than `index (rindex)` when checking for the occurrance of a character in a string.

   (b) Use `memset` for zeroing a block of memory rather than `bzero`.

   (c) Use `memcpy` for copying a block of memory rather than `bcopy`.

   (d) Use `%lf` as the format specification in {`s,f`}`printf` and {`s,f`}`scanf` for doubles. Do not use `%f` for doubles, only for floats.

   (e) Use `%ld` as the format specification in {`s,f`}`printf` and {`s,f`}`scanf` for longs. Do not use `%d` for longs, only for ints.

   (f) Watch out for 'null dereferencing' (indirecting through a pointer that has a zero value). For example:

   ```
   for (; p->bob > 4 && p != 0; p = p->next) {
       /* code */
   }
   ```

   The check for the null pointer should have been made before the use of the pointer. On a VAX, the indirection through a null pointer will not cause an error (and will probably work), but on a SUN the program will exit with a 'Segmentation Violation' (there is a way to get this behavior under ULTRIX if you are interested). The correct form is:

   ```
   for (; p != 0 && p->bob > 4; p = p->next) {
       /* code */
   }
   ```

9. Each tool must have a man page (`$OCTTOOLS/lib/utils/demo.manpage` is an example) and each package must have a doc file (`$OCTTOOLS/lib/utils/demo.doc` is an example) and optionally a man page.

10. The longest pathname (in characters) in your tool directory, including `/src/<toolname>`, must be less than 128 characters. For example:

```
./src/oct2ps/oct2ps.c                          21
./src/misII/decomp/decomp_int.h                31
./src/vulcan/sample/physical/interface;        39
./src/slide-maker/hershey-fonts/showfonts.c    43
```

This reason for this is that the **tar** program requires all pathnames to be less than 128 characters.

11. If you are supported directly or indirectly by the Regents of the University of California you should add the following to each source and header file in your tool/package:

```
#include "copyright.h"
```

This will include the standard copyright notice:

```
#ifndef OCTTOOLS_COPYRIGHT_H
#define OCTTOOLS_COPYRIGHT_H
/*
 * Octtools Distribution 5.2
 *
###############################################################################
#                                                                             #
# Copyright (c) 1988, 1989, 1990, 1991, 1992, 1993                            #
# The Regents of the University of California.  All rights reserved.          #
#                                                                             #
# Permission to use, copy, modify, and distribute this software and its       #
# documentation for any purpose, without fee, and without written agreement   #
# is hereby granted, provided that the above copyright notice and the         #
# following two paragraphs appear in all copies of this software.             #
#                                                                             #
# IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR   #
# DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT #
# OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY   #
# OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.           #
#                                                                             #
# THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,         #
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY    #
# AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS   #
# ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO  #
# PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.      #
#                                                                             #
###############################################################################
 *
 */

#if !defined(lint) && !defined(SABER)
static char octtools_copyright[] =
 "Copyright (c) 1988, 1989, 1990, 1991, 1992, 1993
    Regents of the University of California.  All rights reserved.";
#endif
#endif
```

12. Your tool should conform to the following command line argument standards (some of these flags will not apply to certain tools):

-o to specify an output facet.

cell:view:facet for specifying OCT facets. If any are missing they default to what is appropriate from the context (*i.e.,* if your program works on symbolic views then the view should default to **symbolic**). Use the routines in **oh** to parse this type of strings.

With no arguments or with an incorrect argument the tool should respond with a usage message (automatically done by the **options** package. It is strongly recommended that you use the **options** package for argument list processing.

13. Your `Makefile` must understand a number of targets and variables. See the `Makefile` in the source directory for a description of the available targets and variables. The program **create-octtools-makefile** will automatically make a conformant `Makefile` for your tool or package. Run this program in your source directory, it will ask you some questions, look at your source directory, and then build a `Makefile`.

4.1 Integrating tools with VOV

This library allows tools to leave a trace of their activity. The library is small and non intrusive, in the sense that it does not change the behavior of the tool. If the design manager VOV is not active, all routines return immediately without any action.

These routines give the tool access to the services of the design manager VOV. These services make most sense in the context of a design activity:

- Conflict detection: are the inputs valid? can the outputs be overwritten?
- History tracking.
- Automating retracing of transitions for consistency maintenance.
- Coordination of concurrent activities.
- Others. See other documentation about VOV (e.g. the VOV tutorial).

The following terminology is used:

Transition: a tool invocation.

Place: a piece of data.

Places used by a transition are INPUT places. Places created by a transition are OUTPUT places. Places modified by a transition are both inputs and outputs.

4.1.1 Basic routines

The library consists of 4 basic routines and 5 handy routines:

```
#include "vov.h"

/* Fundamental routines: */
extern int   VOVbegin( int argc, char* argv[] );
extern int   VOVinput ( VovPlaceType type , char* name );
extern int   VOVoutput( VovPlaceType type , char* name );
extern void  VOVend( int status );

/* Other routines: */
int   VOVdelete( VovPlaceType type, char* name );
FILE* VOVfopen( char* filename, char* mode );
int   VOVinputFacet( octObject* facet );
int   VOVoutputFacet ( octObject* facet );
void  _VOVend( int status );
int   VOVdisable();
```

- `int VOVbegin( int argc, char* argv[] );`

  Should be called as soon as possible but preferably just after the tool has decided that it is going to do something. The arguments are those coming from the main(). It returns 0. If VOV is active, a connection with the VOV server is established.

- `void VOVend( int status );`

  This function should be used instead of exit(int status). The function does not return. If VOV is active, the routine informs the server that the tool has completed and closes the connection with the server.

- `int VOVinput ( VovPlaceType type , char* name );`

- `int VOVoutput( VovPlaceType type , char* name );`

  These routines declare that a piece of data is an input or an output for the tool. The most common values for type are `VOV_UNIX_FILE` and `VOV_OCT_FACET`. Please look at `vov.h` for more possible values. The name is the name of the file or the name of the facet, in the form `"cell:view:facet"`. The name does not need to be a full path. The routines return a unique identifier for the place, which can be discarded in most cases.

  If a place is both an input and an output, the input declaration should come first.

  Example: file foo is modified.

  ```
  VOVinput( VOV_UNIX_FILE, "foo" );  /* foo is an input */
  VOVoutput( VOV_UNIX_FILE, "foo" ); /* foo is also an output. */
  ```

  Example: file foo really is just an output.

  ```
    VOVoutput( VOV_UNIX_FILE, "foo" ); /* Foo is an output.*/
    VOVinput( VOV_UNIX_FILE, "foo" );  /* This declaration has no effect,
  because a tool can freely use
  the information it has generated. */
  ```

- `int VOVdelete( VovPlaceType type, char* name )`

  Deletes the named place from the trace. The place is forgotten, as if it had never been declared. Only outputs of the current transition can be deleted. This function is normally used with temporary files: use `VOVoutput()` to declare the temporary file, and then `VOVdelete()` when you are done with the file.

- `FILE* VOVfopen( char* filename, char* mode );`

  A convenient replacement for most fopen() calls. This routine guesses that if mode is "r" the file is an input, if mode is "w" it is an output, and if mode is "a" it is both an input and an output. Nothing happens if the fopen fails. N.B. There are cases in which a file opened with "r" is actually an output for the transition. In those cases, this function should not be used.

- `int VOVinputFacet( octObject* facet );`

- `int VOVoutputFacet ( octObject* facet );`

  These are convenient routines to handle OCT facets. They are equivalent to VOVinput() and VOVoutput().

- `void _VOVend( int status );`

  Same behavior as `VOVend()` except that the routine does not call `exit()`. It can be used by those tools whose operation can be best described as two or more transitions (e.g. VEM, BLIS), depending on what the user decides to do.

- `int VOVdisable();`

  Disables and enables the vov library. It returns 0 if tracing is disabled, 1 if it has been re-enabled. If you use the options package (see doc) you can call this routine with the `-=V` option.

4.1.2 Using VOV

In order for these routines to have any effect, a few conditions have to be met.

1. The environement variables `VOV_HOST_NAME` and `VOV_PROJECT_NAME` have to be set.
2. The program `$OCTTOOLS/bin/vov_channel` must exist.
3. The VOV server must be running.

If 1 and 2 are met, but not 3, the `VOVbegin()` prints an error message, and the library then becomes inactive.

4.2 Sample Programs

This section includes a few of simple OCT programs that should help the novice in writing programs to create analyze and modify OCT facets.

GENERATE.C

```
/*
 * 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 - allocate 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
     */

    /* initialize a generator to return all layers attached to the facet */
    (void) octInitGenContents(&facet, OCT_LAYER_MASK, &lgen);

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

        /*
         * generate over all geometries on the layer
         */

        /* initialize a generator to return all geometries attached to the layer */
        (void) octInitGenContents(&layer, OCT_GEO_MASK, &ggen);

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

            /*
             * process the geometry
             */
        }
```

```
    }

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

    exit(0);
}
```

NETLIST.C

```
/*
 * 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
     */

    /* initialize a generator to return all instances attached to the facet */
    (void) octInitGenContents(&facet, OCT_INSTANCES_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
         */

        /* initialize a generator to return all terminals attached to 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);
}
```

NETLIST.C, using all packages

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

#include "copyright.h"
#include "port.h"
#include "oct.h"
#include "oh.h"
#include "vov.h"
#include "errtrap.h"

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

    VOVbegin(argc,argv);  /* Tell VOV that this tool has started. */
    octBegin();           /* initialize oct */

    /*
     * open the facet to netlist
     */
    if( ohOpenFacet(&facet,argv[1],argv[2],"contents","r" ) < OCT_OK ) {
        errRaise("netlist", 1, "Could not open %s.\n\t%s",
         ohFormatName(&facet), octErrorString() );
    }
    VOVinputFacet(&facet);  /* Declare facet as an input. */

    /*
     * generate over all instances
     */

    /* initialize a generator to return all instances attached to the facet */
    OH_ASSERT(octInitGenContents(&facet, OCT_INSTANCES_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
         */

        /* initialize a generator to return all terminals attached to the instance */
        OH_ASSERT(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();

    VOVend(0);       /* Transition has ended. Notify VOV of exit status */
}
```

Chapter 5

RPC: Remote Procedure Call

A goal of VEM and OCT was to make it work in our distributed environment, one of multiple machines (of different types and operating systems) and multiple languages. OCT works in this distributed environment by using a remote file system to access the database, thus allowing the database to be spread out over many machines. VEM works by using a network-transparent windowing system which allows the I/O devices (display, keyboard, mouse) and the program controlling them to be on different machines. VEM and user applications communicate in this environment by using remote procedure calls.

The Remote Procedure Call (RPC) package allows user applications to run as separate processes outside of the VEM address space. The applications make subroutine calls to VEM and OCT as if they were in the same process and address space, similar to tightly-bound VEM commands. The RPC client, that is the RPC code linked with the application program, interprets these calls and passes them to VEM. The RPC server, linked with VEM, calls the appropriate VEM and OCT routines, and returns the results to the RPC client. Having user applications as separate processes communicating with VEM via a local-area network using RPC has several important advantages:

CPU Servers and Custom Hardware: Applications that are computationally expensive and require CPU servers (*e.g.,*CRAY X/MP, VAX 8800, IBM 3081) or that require special hardware (*e.g.,*parallel processors, LISP machines) can be run as a separate process communicating with VEM over a local-area network.

Different Languages: Since applications run outside of the VEM address space, a full range of implementation language choices are available. This includes languages other than the one that VEM is written in, and in fact applications are not even limited to the languages that run on the same machine as VEM. Remote applications for our system have been written in C, FranzLisp, and VAXLISP.

Asynchronous, Separate Processes: The remote applications are asynchronous, thus they do not tie up VEM (as in tightly-bound VEM commands, which block VEM until they are finished). Also, if the remote application dies, the VEM process can continue, thus a certain amount of protection is provided. By use of OCT change lists, VEM can determine that a remote application has changed the database and automatically update the display. Since the applications are separate processes, more than one can be running from a single VEM and the underlying operating system can deal with scheduling.

Portability: Since RPC is a protocol and procedure-call interface, it is independent of any particular implementation language. Two RPC application libraries have been written, one in C and the other in a combination of C and VAXLISP.

Invisibility: The RPC package provides an interface to the application writer that looks exactly the same as if the application writer were directly linking with VEM, thus all argument passing to the remote function and calls to VEM and OCT are handled transparently.

No Editor Source Changes: The VEM bindings file defines where to find a remote application (host and file system location). This file also registers menu entries, key bindings, and type-in commands for invoking remote applications. Therefore no changes have to be made to VEM to add a new remote application; this is in contrast to tightly-bound commands that require some recompilation and relinking.

5.1 How RPC Applications Work

The user communicates with the remote application by building VEM argument lists and requesting that a remote function be executed. This appears exactly the same as that of a tightly-bound VEM command. However, control is immediately returned to the user without waiting for the remote command to finish. Remote functions also look like the tightly-bound VEM functions at the programming level. The remote application is started using an operating system remote execution function. The application then connects back to VEM via a reliable byte stream, and sends a list of commands and remote function bindings. Once the remote menu is registered, a user-written initialization function sets up the remote application before entering a loop waiting for remote function requests from VEM. If the remote application is batch-oriented, it may not register any remote functions and the user initialization function may be where all computations for the application are done.

The RPC package is specific to our system of remote functions communicating with VEM and OCT; so the protocol and interface are specific to VEM and OCT. This is similar to X (as a matter of fact, our decision to use remote procedure calls was influenced by our experience with X[6]), where the interface and protocol are specific to the X model of bitmapped graphics. There are other remote procedure call protocols and packages that are more general, such as SUNRPC[5] and Courier[1], but they sacrifice some efficiency for this generality. The RPC package can minimize the amount of data passed to and from the remote application by knowing what the contents of the data items passed are and what they are used for. A common data item, the OCT object, is used in every OCT call, but in many of the calls only a small part of the object is used and thus a smaller amount of data can be sent. Most, but not all, of the RPC calls from the application to VEM are synchronous, that is they wait for VEM to respond; this is in contrast to the asynchronous X window system, where remote clients (such as VEM) do not wait for the server to send back a response for most functions. Note that applications are asynchronous with respect to VEM, which does not wait for remote applications to finish. However, applications are synchronous with respect to most of the individual VEM and OCT requests they make, the remote applications wait for VEM to process the request and return the result.

5.2 RPC Requirements

There are a few requirements imposed by RPC on the system running the graphics editor, the system running the application and the transport medium.

The RPC server (VEM) and RPC client (user application) must be able to make a reliable byte stream connection for sending remote function requests, data and return codes. Both TCP-IP[?] and DECNET[?, ?] have this ability. VEM must be able to start the execution of the application on a remote machine. The `rsh` command and `rshd` daemon of 4.3 BSD, HPUX (with 4.3 BSD networking option) and ULTRIXare examples of this. The operating system that VEM runs under must support some means of determining if a connection has data to be read. The *select* system call of 4.3 BSD, HPUX (with the 4.3 BSD networking option) and ULTRIX is an example of this. The RPC server and client must have a `long` integer datatype that is atleast as big as a pointer, all pointers must be the same size, and a `double` datatype must fit in two longs.

5.3 Current RPC Applications

The following are some of the RPC applications that are currently being used:

symhelp: a checker for the oct sumbolic policy.

steps: An interface between symbolic and schematic to SPICE3.

slide-maker: a C-based program for creating and editing color slides (for 35mm slides or overhead transparencies).

blimp: a C-based IC lab monitoring program.

tech-edit: a C-based technology facet editor.

nle: C-based programs for displaying routing and channel information.

atlas: C-based program for defining routing channels.

gabriel: a Franz LISP-based program for Z-domain simulation

5.4 Using RPC Applications

Since RPC applications are run from VEM, the first step is to set up VEM so the remote application can be run (see Section 5.4.1). Once this is done, you must set up your remote environment so the remote application can be run (see Section 5.4.2). Once VEM and the remote environment are set up, VEM can be run and a remote application can be invoked in the same manner as any other VEM command (invoked via the menu, or if specified in the `vem.bindings` file by single keystroke or typing the command name).

Only one remote application can be running at a time in a window (and its' subwindows). Any number of remote applications can be running from a single VEM session, but must be running in different windows.

Only one remote user function can be running at a time for each remote application. The RPC system locks out any remote user functions until the current one is complete. Any number of remote user functions can be running at the same time, but only one per application.

5.4.1 VEM Bindings File

For each remote application that you want to be able to run, the following lines must be in your `vem.bindings`.

REMOTE	name	host[/user[/password]][:protocol]			path
viewType	name	**menu**	paneName	selectionNumber	**alias**

name: name to appear in the menu.

host: name of the host to run the remote application on. If **localhost**, then the remote application is run on the local machine. **host** can not have any "/" characters in it.

user: name of the user to run the remote application as. The default is the name of the local user. **user** can not have any ":" characters in it.

protocol: name of the protocol to use for transmitting data between VEM and the remote application. Can be one of **inet** (TCP-IP), or **decnet**. The default is **inet**.

path: path name of the remote application on the remote machine. The path name is relative to the home directory of the remote user. "~", ".", and ".." can be used in the path name.

viewType: name of the view specific menu that the application is to appear in. Currently VEM supports **schematic**, **symbolic** and **physical**.

paneName: name of menu pane where the remote application **name** is to appear. If a menu entry is not wanted for the remote application, leave out **menu**, **paneName**, and **selectionNumber**.

selectionNumber: location on the menu pane. The first entry on a pane is 0.

ALIAS: if specified, allows the user to type in the name of the remote application in addition to selecting it from the menu.

For example:

REMOTE	show-ultrix	eros	$OCTTOOLS/demo/rpc/show
REMOTE	show-ultrix-dn	EROS:decnet	$OCTTOOLS/demo/rpc/show
REMOTE	show-sun	seguaro/davidh	~davidh/rpc/show
INCLUDESTD			

symbolic	show-ultrix	MENU	Applications	0	ALIAS
symbolic	show-ultrix-dn	MENU	Applications	1	ALIAS
symbolic	show-sun	MENU	Applications	3	ALIAS

RPC-ANY

VEM has a command, **rpc-any**, that can be used to run remote applications that are not in the VEM bindings file.

```
[args] "host path" : rpc-any
```

Host and **path** are the same as in the bindings file and **[args]** is an optional list of VEM arguments that is sent to the remote application. If "host" is left out, if will default to "localhost".

RPC_RESET

VEM has a command, **rpc-reset**, that can be used to reset the flag that tells VEM that an application is running in a window. This is useful for those times when an application is started but dies because of "Permission Denied" or "File Not Found".

5.4.2 .rhosts file

Since RPC uses `rsh` to start remote applications, you must be in the `.rhosts` file on the remote machine. **Even if you are attempting to execute the remote application on your local machine, you need to be in your local `.rhosts` file.**

5.4.3 Random Stuff

Application output to standard output and standard error will go to the window where VEM was started. Remote applications can be killed from VEM via the tightly bound command **`killAppCmd`.** This command kills the application running in the window containing the mouse. Note, if the remote application is "setuid" to somebody other than you and you are not "root", forget about this working.

5.5 Programming Interface

A remote application is composed of a set of remote functions that the VEM user can invoke via VEM menu-selected and type-in commands, and a programmer-written initialization function, `UserMain`.

5.5.1 UserMain

`UserMain` does any initialization that must be done before any of the user functions in the application menu can be invoked. In some remote applications that do not wish to register a menu, this may be where all execution is done. Executing `RPCExit(status)` will cause the remote application to end and VEM will notify the user.

```
long
UserMain(display, spot, cmdList, uow, menuArray)
char *display;        /* name of the X display            */
RPCSpot *spot;        /* window, facet and mouse location */
lsList cmdList;       /* VEM argument list                */
long uow;  /* for future use                    */
RPCFunction **menuArray; /* array of menu entries         */
```

Before returning, `UserMain` must fill in `menuArray` with the remote commands array (correspondences between the remote user functions and the names on the menu). `UserMain` then returns the number of items in the array; 0 if there are no remote commands to be registered.

The command array has the following form:

```
struct RPCFunction {
    rpcStatus (*function)(); /* user function to call       */
    char *paneString;        /* name of the menu pane       */
    char *menuString;        /* name of the menu selection  */
    char *keyBinding;        /* keybinding for the command  */
    long userOptionWord;  /* Value passed to the command */
}
```

`paneString` and `menuString` can not contain spaces. An entire pane must be described before the next one can be started. The first array is correct, the second is in error:

```
    RPCFunction CommandArray[] = {
         {criticRun,  "General", "Run",    "r", 0},
         {criticExit, "General", "Exit",   "e", 0},
         {test1,      "Test",    "Test-1",  0,  0}
         {test2,      "Test",    "Test-2", "T", 0}
    };

    RPCFunction CommandArray[] = {
         {criticRun,  "General", "Run", "r", 0},
         {test1,      "Test",    "Test-1", 0, 0}
         {criticExit, "General", "Exit", "e", 0},
         {test2,      "Test",    "Test-2", "T", 0}
    };
```

If the command array is static (as in the above example), then the following simple `UserMain` would register the commands.

```
UserMain(display, spot, arglist, uow, array)
char *display;
RPCSpot *spot;
lsList arglist;
long uow;
RPCFunction **array;
{
    /* normal user main stuff */
```

```
        /* set up the command array */
        *array = CommandArray;
        return(sizeof(CommandArray)/sizeof(struct RPCFunction));
    }
```

Since the command array is set in `UserMain`, the application can create a menu based on the arguments to `UserMain` and the OCT database.

5.5.2 Remote User Functions

User functions are invoked by selecting the corresponding menu string on the application menu (or by typing the string). The user functions take three arguments: `spot`, `cmdList` and `uow`. The user functions must return `RPC_OK`.

```
    rpcStatus
    remoteFunction(spot, cmdList, uow)
    RPCSpot *spot;         /* window, facet and mouse location  */
    lsList cmdList;        /* VEM argument list - of (RPCArg *) */
    long uow;              /* from the command array            */
```

`Spot` contains the window that the command was invoked in, the OCT Id of the facet that is displayed in the window, and the location in OCT coordinates where the mouse was when the command was invoked.

```
    struct RPCSpot {
        Window theWin;
        octId facet;
        struct Point thePoint;
    };
```

`cmdList` is the VEM argument list built up by the user (text, points, lines, boxes, and objects).

```
    struct RPCPointArg {
        Window theWin;
        octId facet;
        struct octPoint thePoint;
    }

    struct RPCObjArg {
     octId facet;
        octId theBag;
        struct octTransform tran;
    }

    struct RPCArg {
        long argType;                    /* type of argument    */
        long length;                     /* length of argument  */
        union {
            struct RPCPointArg *points;  /* array of OCT points  */
            char *string;                /* text string          */
            struct RPCObjArg objArg;     /* Id for OCT_BAG       */
        } argData;
    };
```

Argument types are: `VEM_POINT_ARG`, `VEM_BOX_ARG`, `VEM_LINE_ARG`, `VEM_TEXT_ARG`, and `VEM_OBJ_ARG`. The length field is the number of points in the `points` array for points and lines, and the number of pairs of points in the `points` array for boxes. Executing `RPCExit(status)` in a remote function will cause the remote application to end and VEM will remove the remote menu and notify the user.

List Package

List manipulation routines are available to use with the VEM argument lists. For a description of the package and the routines available, see:

`$OCTTOOLS/doc/list.doc`

OCT Package

All OCT calls, objects and properties are available to the remote application. `octBegin` and `octEnd` are automatically called. More information on OCT can be found in:

`$OCTTOOLS/doc/oct.me`

There are 2 *special* OCT calls that speed up the generation of objects: `octInitGenContentsSpecial` and `octInitGenContainersSpecial`. If you will not modify any of the objects that have not been, but will be generated by the generator, you can start the generator with `octInitGenContentsSpecial` or `octInitGenContainersSpecial`. These functions create generators that send objects to the remote application in large blocks rather than one object at a time. As an example, generating objects can be sped up by more than a factor of 10 by using the *special* calls.

Region Package

The region package is used for region searches in an OCT facet. The procedure calls and data structures for this package are exported via RPC. For a description of the package, see:

`$OCTTOOLS/doc/region.doc`

VEM Package

The following VEM calls are available to the remote application. The VEM package and dialog manager package routines return `vemStatus`:

vemStatus	meaning
VEM_OK	everything is ok
VEM_NOMORE	no more items
VEM_CANTFIND	can not find the object
VEM_NOTUNIQ	the object in not unique
VEM_NOSELECT	no items selected (abort requested)
VEM_CORRUPT	something wrong
VEM_TYPECONF	illegal parameter value

```
vemStatus vemMessage(message, options)

char   *message;   /* message to display and/or log   */
int    options;    /* where to send the message       */
```

Sends `message` to the VEM console. `options` tells where to send the message.

option	message destination
0	VEM log file
MSG_DISP	VEM console window and VEM log file
MSG_DISP|MSG_NOLOG	VEM console window only

If the message string contains `control-F` followed by 0, 1, or 2, the font and/or color of the text will change (in the VEM console window). `0` gives the font specified by the `norm` font specification in VEM, `1` gives the font specified by the `bold` font specification, and `2` gives the font specified by the `italic` font specification.

For convenience, the macro `VEMMSG(str)` has been defined that sends `str` via `vemMessage` to the VEM console windoow.

```
void vemPrompt()
```

Display a VEM prompt in the console window and redisplay the argument list.

```
vemStatus vemCommand(command, spot, cmdList, uow)
char     *command;  /* VEM command to execute           */
RPCSpot  *spot;     /* window, facet and mouse location  */
lsList   cmdList;   /* VEM argument list                */
long     uow;       /* ignored                          */
```

Calls an arbitrary VEM command (such as *pan*, *zoom* or *close*). NOTE: make sure you set all of the fields in spot and the arglist, if not, strange errors can occur!

```
vemStatus vemWnQRegion(facet, box)
octObject      *facet;  /* facet to update   */
struct octBox  *box;    /* region to update  */
```

Tell VEM to queue the region in the box for redraw.

```
vemStatus vemWnFlush()
```

Causes a screen update (if any screen regions have been queued for redisplay).

```
vemStatus vemWnGetOptions(window, options)
Window   window;    /* window to get the options for  */
wnOpts   *options;  /* options structure              */
```

Get the window options structure for a window. The window options structure is defined as:

```
typedef struct win_options {
    int disp_options; /* Display option bits (window.h)    */
    int lambda; /* Oct units per lambda              */
    int snap; /* Input coordinates snapped to this */
    /* Grid related options */
    int grid_base; /* Logarithmic base of grid lines    */
    int grid_minbase; /* Minimum base of logarithmic grid  */
    int grid_majorunits; /* Major grid line frequency         */
    int grid_minorunits; /* Minor grid line frequency         */
    int grid_diff; /* Difference before grid not displayed */
    /* Display related options */
    double bb_thres; /* Bounding box name threshold       */
    int bb_min; /* Abstraction level of bounding boxes */
    int con_min; /* Abstraction level of contacts     */
    int solid_thres; /* Objects drawn solid at this level */
```

```
    char *interface; /* Interface facet to display        */
    int inst_prio; /* Instance priority level           */
} wnOpts;

/* field names for 'disp_options' */

#define VEM_GRIDLINES 0x001 /* Grid lines drawn (if visible)          */
#define VEM_TITLEDISP 0x002 /* Display window title                   */
#define VEM_EXPAND 0x004 /* Expand instances in this window        */

/* read-only field */
#define VEM_REMOTERUN 0x008 /* Remote application running in this window*/

#define VEM_SHOWDIFF 0x010 /* Show difference coordinates             */
#define VEM_SHOWABS 0x020 /* Show absolute coordinates               */
#define VEM_SHOWBITS 0x040 /* Show status bits                        */
#define VEM_SHOWEDIT 0x080 /* Show edit cell name                     */
#define VEM_SHOWCON 0x100 /* Show context cell name                   */
#define VEM_SHOWBB 0x200 /* Show instance bounding boxes             */
#define VEM_SHOWRPC     0x400   /* Show RPC application name              */

/* read-only field */
#define VEM_NODRAW 0x800 /* Make window inactive                     */
#define VEM_MANLINE 0x1000 /* Manhattan line arguments                */
#define VEM_GRAVITY 0x2000 /* Argument gravity for symbolic/schematic */
#define VEM_SHOWAT 0x4000 /* Show actual terminals   */
#define VEM_DOTGRID 0x8000 /* Display dotted grid   */
```

```
vemStatus vemWnSetOptions(window, options)
Window   window;    /* window to set the options for  */
wnOpts   *options;  /* options structure              */
```

Set the window options structure for a window. Note that the fields named `VEM_REMOTERUN` and `VEM_NODRAW` in `disp_options` are read-only.

```
vemStatus vemWnTurnOffLayer(window, layerName)
Window   window;      /* window to set the options for  */
char     *layerName;  /* name of the layer to turn off  */
```

Turn off display of the layer named `layerName` in the window.

```
vemStatus vemWnTurnOnLayer(window, layerName)
Window   window;      /* window to set the options for  */
char     *layerName;  /* name of the layer to turn on   */
```

Turn on display of the layer named `layerName` in the window.

```
Window vemOpenWindow(facet)
octObject  *facet;  /* facet to open the window on  */
char       *geo;    /* X geometry specification     */
```

Open a VEM window on `facet`, returns a window handle.

```
void vemFlushTechInfo(facet)
octObject  *facet;  /* facet to use to find the technology  */
```

Flush the technology associated with a particular facet.

```
void vemSetDisplayType(name, chromatism)
char  *name;       /* name of the display        */
int   chromatism;  /* chromatism of the display  */
```

Set the name and chromatism of the display.

```
void vemGetDisplayType(name, chromatism)
char  **name;       /* name of the display        */
int   *chromatism;  /* chromatism of the display  */
```

Get the name and chromatism of the display.

```
vemSelSet vemNewSelSet(id, red, green, blue, len, style, width, height, pat)
octId           id;                /* id of containing object - usually facet  */
unsigned short  red, green, blue;  /* display color of the selected set        */
int             len, style;        /* type of line                             */
int             width, height;     /* type of line                             */
char            *pat;              /* fill pattern                             */
```

This routines creates a selected set that displays with a fill pattern. The range of `red`, `green`, and `blue` is 0 to 65535. The table shows how `len` and `style` affect the the bounding box lines. `fillpat` is a string consisting of `height` groups of `width` `0` or `1` characters with the groups separated by spaces, tabs or newlines. For example, `width` and `height` equal to 8 with `pat` equal to "00000000 00000000 00000000 00011000 00011000 00000000 00000000 00000000" will create a sparse dotted pattern, like vem uses when highlighting objects. A `width` and `height` of 1 with `pat` equal to "1" will create a solid fill, and with `pat` equal to "0" will create an outline.

Line Type	Len	Style	Binary
solid line	1	1	1
	2	3	11
dotted line	2	1	01
	4	5	0101
dot-dashed line	5	11	01011
dashed line	6	27	011011

Note: the function will return `(vemSelSet) 0` on failure.

```
vemStatus vemClearSelSet(selset)
vemSelSet  selset;  /* selected set to clear  */
```

Remove all items from a selected set.

```
vemStatus vemAddSelSet(selset, objectId)
vemSelSet  selset;    /* selected set to add to  */
octId      objectId;  /* object to be added      */
```

Add an object and all objects transitively contained by it to a selected set. Note that all of the geometric objects added to a selected set, either via direct addition or addition via containment, must have a bounding box; therefore, make sure any paths or polygons have points associated with them before adding them to a selected set.

```
vemStatus vemDelSelSet(selset, objectId)
vemSelSet  selset;    /* selected set to delete from  */
octId      objectId;  /* object to be deleted         */
```

Delete an object and all objects transitively contained by it from a selected set.

```
vemStatus vemZoomSelSet(selset, window, scale)
vemSelset  selset;  /* selected to zoom to  */
Window     window;  /* window to zoom in    */
double     scale;   /* scale                */
```

Zoom a window to the selected set. Scale of 1.0 zooms to the bounding box of the selset, 2.0 zooms to twice the size of the bounding box.

```
vemStatus vemFreeSelSet(selset)
vemSelSet  selset;  /* selected set to free  */
```

Free the storage for a selected set. It is not necessary to call `vemClearSelSet` before calling `vemFreeSelSet`.

```
vemStatus vuFindSpot(spot, rtnObj, mask)
RPCSpot        *spot;    /* cursor location          */
octObject      *rtnObj;  /* object under the cursor  */
octObjectMask  mask;     /* object mask              */
```

This routine looks for geometry under the given `spot`. If there is a unique object there, it is returned in `rtnObj`. If it is ambiguous, the user is queried using a dialog box to choose one. `mask` can be any combination of `OCT_INSTANCE_MASK`, `OCT_GEO_MASK`, `OCT_TERM_MASK`. If the object is an instance and `OCT_TERM_MASK` is in `mask`, it also looks to see if the point is over an actual or formal terminal. If so, the terminals are added to the list of items presented to the user. If there is no object under the spot, it returns `VEM_CANTFIND` and rtnObj is set to the spot facet. If no selection is made, it returns `VEM_NOSELECT` and `rtnObj` is unmodified.

```
vemStatus vuFindLayer(facet, spot, layerName)
octObject        *facet;      /* oct facet        */
struct octPoint  *spot;       /* cursor location  */
char             *layerName;  /* layer name       */
```

This routine looks for a layer under `spot`. If the layer is unique, the name is returned in `layerName`. If it is ambiguous, the user is queried using a dialog box to choose one.

Dialog Manager Package

The following functions and structures are used to display simple dialog boxes.

The following structures are used by the Dialog Manager Package:

```
typedef struct {
    char *itemName;       /* displayed name                */
    Pointer userData;     /* optional associated user data */
    int flag;             /* non-zero if user selected it  */
} dmWhichItem;

typedef struct {
    char *itemPrompt;     /* type-in prompt                */
    int rows, col;        /* size of type-in               */
    char *value;          /* optional default value        */
    Pointer userData;     /* optional associated user data */
} dmTextItem;
```

```
vemStatus dmMultiWhich(title, count, items, itemFunc, help_buf)
```

```
char         *title;         /* question at the top of the dialog  */
int          count;          /* number of items                    */
dmWhichItem  *items;         /* array of items                     */
int          (*itemFunc)();  /* ignored                            */
char         *help_buf;      /* help string                        */
```

This routine presents a *moded*[1] dialog which asks the user to select zero or more items from the list provided. The dialog consists of a title, an array of strings inside the buttons, and the two buttons **Done** and **Abort**. Once the user presses the **Done** button, the selected items will be marked using the `flag` field of the `dmWhichItem` structure. In the RPC version, `itemFunc` is ignored. Pressing the `Abort` button causes the function to return `VEM_NOSELECT`. If `help_buf` is not zero, a help button will be displayed and if it is pressed, a help window containing the contents of `help_buf` will appear.

```
vemStatus dmWhichOne(title, count, items, itemSelect, itemFunc, help_buf)
```

```
char         *title;         /* question at the top of the dialog  */
int          count;          /* number of items                    */
dmWhichItem  *items;         /* array of items                     */
int          *itemSelect;    /* selected item                      */
int          (*itemFunc)();  /* ignored                            */
char         *help_buf;      /* help string                        */
```

This routine presents a *moded* dialog which asks the user to select exactly one item from the array provided. The dialog consists of a title, an **Abort** button, and an array of strings each with a blender control. If a selection is made, the routine will return `VEM_OK` and set `itemSelect` to the appropriate number. If the selection is aborted, the routine will return `VEM_NOSELECT` and `itemSelect` will be set to -1. In the RPC version, `itemFunc` is ignored. If `help_buf` is not zero, a help button will be displayed and if it is pressed, a help window containing the contents of `help_buf` will appear.

```
vemStatus dmMultiText(title, count, items)
```

```
char         *title;  /* question at the top of the dialog  */
int          count;   /* number of items                    */
dmTextItem   *items;  /* array of items                     */
```

[1] freezes the server until sent away

This routine presents a *moded* dialog which asks the user to fill in a number of fields with text. The dialog consists of a title, an **OK** button, and **ABORT** button, and an array of items with specify the prompt, size of the *type-in* field, default value for the *type-in*, and optional user data. The user is expected to move between fields with tabs filling in the fields until he is satisfied, then then activate the **OK** button. In this case, the routine returns `VEM_OK` and sets the `value` field in each item to the typed in values. This string is owned by the dialog package and must not be disturbed! If the user does not want to accept the values, the **ABORT** button can be activated and the routine returns `VEM_NOSELECT`.

```
vemStatus dmConfirm(title, message, yesButton, noButton)
char  *title;      /* title at the top of the dialog  */
char  *message;    /* explanation message             */
char  *yesButton;  /* yes button label                */
char  *noButton;   /* no button label                 */
```

This routine presents a *moded* dialog which asks the user whether or not he wishes to continue. The dialog consists of a title, explanation text, and two buttons. If the user hits the *yes* button, the routine returns `VEM_OK`. Otherwise, it returns `VEM_FALSE`. It `noButton` may be empty (null) in which case only the *yes* button will be shown.

RPC Package

```
char *RPCVersionString()
```

Return the RPC version string.

```
void RPCExit(status)
int  status;  /* status code to be returned to the user  */
```

Exit the remote application cleanly. `exit()` can be used, but `RPCExit()` is the preferred way to exit an application.

```
char *RPCXGetDefault(program, name)
char  *program;  /* name of the program  */
char  *name;     /* name of the default  */
```

Get an X default from the VEM side. `Program` must be "vem" and `name` should be "remoteApplicationName.item".

```
FILE *RPCfopen(file, mode)
char  *file;  /* file to open      */
char  *mode;  /* mode of the open  */
```

This routine is used to open files relative to the current working directory of VEM. Since the remote application runs relative the the home directory of the remote user on the remote machine, `fopen` would create the file from that position; this is rarely what is wanted. The arguments to `RPCfopen` are exactly the same as `fopen`. At this time, only **write** and **append** are allowed.

The following functions are hardly ever used, but are needed if you wish to control X windows, or do other similar activities, from a remote application.

```
void RPCUserIO(length, *readmask, *writemask, *exceptmask, function)
int    *readmask;       /* bit array of file descriptors  */
int    *writemask;      /* bit array of file descriptors  */
int    *exceptmask;     /* bit array of file descriptors  */
void   (*function)();   /* function to call               */
```

Register the selection masks with the RPC package. This call replaces the effects of any previous calls. If any of the file descriptors are selected on, `function` is called as:

```
(*function)(nselected, readmask, writemask, exceptmask)
int nselected;
int *readmask;
int *writemask;
int *exceptmask;
```

This function is useful for user controlled X[?] applications. Note that there is a limit of 32 on `length`. See *select* (section 2 of [?]) for more information on this.

IMPORTANT: the file descriptors are only selected on when the remote application is *idle*; the file descriptors will not be looked at while a remote function is executing, thus X window exposure events will not be seem until the system returns to its idle state. If you wish handle events while not in the *idle* state, see `RPCsigio` below.

```
void RPCsigio(fd, function)
int   fd;              /* file descriptor   */
int   (*function)();   /* function to call  */
```

This tells the system that if there is any data to be read on the file descriptor `fd`, the process should be interrupted and `function` should be called. When `function` returns, the process will continue from the point of interruption. Any X[?] operations done in your code after a call to `RPCsigio` must be surrounded by `CRITICAL(...)`. `CRITICAL` turns off interrupts during execution of the code inside of it. For example:

```
CRITICAL(
    XText(....);
    XFlush(...);
)
```

```
void RPCRegisterDemon(changeList, function)
octObject   changeList;                   /* change list that triggers the demon   */
void        (*function)(octObject* cl);   /* fuction called                        */
```

Register an RPC demon controlled by changeList. Whenever changeRecords are attached to the changeList, function func will be executed by the client. The demon is invoked only once. If multiple invocations are desired, another changeList must be created and the demon must be re-registered. Following is an example code fragment:

```
void changesProcess( changeList )
    octObject *changeList;
{
```

```
    octObject cr, inst, facet;
    octGenerator gen;

    octGetFacet( changeList, &facet );
    OH_ASSERT(octInitGenContents( changeList, OCT_CHANGE_RECORD_MASK, &gen ));
    while ( octGenerate( &gen, &cr ) == OCT_OK ) {
if ( octGetByExternalId( &facet, cr.contents.changeRecord.objectExternalId, &inst ) == OCT_OK ) {
    fprintf( stderr, "Detected instance creation %s\n", ohFormatName( &inst ) );
    /* Do something ... */
}
    }
    octDelete(changeList); /* Delete old change list */
    changesStart( &facet );     /* Re-register the demon */
}

changesStart( facet )
    octObject *facet;
{
    octObject changeList;
    changeList.type = OCT_CHANGE_LIST;
    changeList.contents.changeList.objectMask = OCT_INSTANCE_MASK;
    changeList.contents.changeList.functionMask = OCT_CREATE_MASK;
    OH_ASSERT(octCreate( facet, &changeList ));
    RPCRegisterDemon( &changeList, changesProcess );
}
```

5.6 Putting Together a Remote Application

5.6.1 Include Files

Two include files are required for remote applications and two can optionally be used. Remote application source files that use any of the data structures or functions described in the program section (Section 5.5) must have the following:

```
    #include "port.h"
    #include "rpc.h"
```

5.6.2 Linking Remote Applications

ULTRIX, BSD, and SUN

To create a remote application for ULTRIX, 4.3 BSD, or SUN:

`cc -Iinc_dir -o binary sources LIBS -lm`

`inc_dir` is where `rpc.h` is kept (usually ~octtools/include).

`binary` is the name of the remote application.

`sources` is a space separated list of source or object file names.

`LIBS` is a list of libraries need for rpc (rpc, list, st, tr, utility, errtrap, uprintf, port). Note that if you use **create-octtools-makefile** the link lines will automatically be generated for you.

```
cc -I/usr/octtools -o gentest gentest.c -L/usr/octtools
       librpc.a liblist.a libst.a libtr.a libutility.a
       liberrtrap.a libuprintf.a libport.a -lm
```

5.6.3 Debugging Remote Applications

For ULTRIX and 4.3 BSD systems remote applications can be debugged with `dbx` by using the remote procedure **rpc-dbx**. **Rpc-dbx** is invoked as a remote application, this starts up an `xterm` with `dbx` running in it. The user is prompted for a binary to run (the remote application) and then can enter `use` and `stop` commands. The remote application can then be started by entering `runrpc`.

Lint Library

There is a lint library, `llib-lrpc.ln`, that can be used for linting a remote application against.

Error Messages

There are a number of RPC error messages that can be returned from the RPC library and VEM.

RPC Error: bad host

Remote machine does not know about the host.

RPC Error: protocol not supported

The protocol in the `vem.bindings` file is not available in the RPC library. Probably used **decnet** with the local or remote hosts not being a VAX or the RPC library was not built with `-DDNET`.

RPC Error: error in user function

A remote user function did not return `RPC_OK`.

RPC Error: incorrect number of arguments to remote application

You are using an old version of the RPC library; relink with a new version of `librpc.a`.

RPC Error: illegal oct object type

RPC Error: illegal property type

Probably trying to pass an OCT object rather than a pointer to an OCT object to an RPC function.

RPC Error: bad vem arg type

The type of an `RPCArg` is not one of: `VEM_POINT_ARG`, `VEM_LINE_ARG`, `VEM_BOX_ARG`, `VEM_TEXT_ARG`, or `VEM_OBJ_ARG`.

RPC Error: bad file mode

Used a file mode in `RPCfopen` that is not supported.

5.7 Example RPC Applications

This section contains three example RPC applications. The first is a very simple one with no remote user functions; the second tries to use as many of the features of RPC as possible; and the third shows how RPC applications can be debugged.

```
#include "copyright.h"
#include "port.h"
#include <sys/types.h>
#include <sys/timeb.h>
```

```
#include "rpc.h"

char *optProgName = "gentest";

/*ARGSUSED*/
int
UserMain(display, spot, arglist, uow, array)
char *display;
struct RPCSpot *spot;
lsList arglist;
long uow;
RPCFunction **array;
{
    octObject facet;
    octGenerator gen;
    octObject object;
    struct timeb start, end;
    char buffer[1024];
    double elapsedTime;

    long count = 0;

    facet.objectId = spot->facet;

    ftime(&start);

    (void) octInitGenContentsSpecial(&facet, OCT_ALL_MASK, &gen);

    while(octGenerate(&gen, &object) == OCT_OK) {
count++;
    }

    ftime(&end);

    sprintf(buffer, "starting time is %f\n",
    (double) start.time + (double) start.millitm / 1000.0);
    VEMMSG(buffer);
    sprintf(buffer, "%d items\n", count);
    VEMMSG(buffer);
    sprintf(buffer, "end time is %f\n",
    (double) end.time + (double) end.millitm / 1000.0);
    VEMMSG(buffer);
    elapsedTime = (double) (end.time - start.time)
      + (double) (end.millitm - start.millitm) / 1000.0;

    if (count > 0) {
sprintf(buffer, "delta time is %f\ntime per object is %f\n",
elapsedTime, elapsedTime / (double) count);
VEMMSG(buffer);
    }

    RPCExit(0);
}
```

```
#include "copyright.h"
#include "port.h"
#include <X/Xlib.h> /* not needed for normal RPC application */
#include "rpc.h"

char *optProgName = "testrpc";

/*
 * variables used between remote functions
 *   not needed for normal RPC application
 */
vemSelSet set = 0;
Font theFont;
char *text = "testrpc user X", *font;

char ubuffer[2048];

#define MYNIL(type)  (type *) 0

char helpString[] = "This is a help message for use in a dialog box.\n\
It is used to explain the purpose of the dialog\n\
box and what options the user has."

#define MIN(a,b)     ((a) < (b) ? (a) : (b))
#define MAX(a,b)     ((a) > (b) ? (a) : (b))

/*ARGSUSED*/
int
testOpen(spot, cmdList, userOptionWord)
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
{
    octObject facet;

    facet.objectId = spot->facet;
    if (octGetById(&facet) < OCT_OK) {
        octError("get by id on facet");
RPCExit(-1);
    }

    (void) sprintf(ubuffer, "UOW is %d\n", userOptionWord);
    VEMMSG(ubuffer);

    (void) vemOpenWindow(&facet, MYNIL(char));

    vemPrompt();

    return(RPC_OK);
}
```

```
/*ARGSUSED*/
int
testVemMessage(spot, cmdList, userOptionWord)
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
{
    RPCArg *arg;
    Pointer pointer;


    if (lsLength(cmdList) != 1) {
VEMMSG("\"string\" vem-message\n");
vemPrompt();
return(RPC_OK);
    }

    (void) sprintf(ubuffer, "UOW is %d\n", userOptionWord);
    VEMMSG(ubuffer);

    (void) lsFirstItem(cmdList, &pointer, (lsHandle *) 0);
    arg = (RPCArg *) pointer;

    if (arg->argType != VEM_TEXT_ARG) {
VEMMSG("\"string\" vem-message\n");
vemPrompt();
return(RPC_OK);
    }
    VEMMSG("testrpc: ");
    VEMMSG(arg->argData.string);
    VEMMSG("\n");

    vemPrompt();

    return(RPC_OK);
}


int
testVemCommand(spot, cmdList, userOptionWord)
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
{
    RPCArg *arg;
    Pointer pointer;


    (void) lsDelEnd(cmdList, &pointer);
    arg = (RPCArg *) pointer;

    (void) sprintf(ubuffer, "UOW is %d\n", userOptionWord);
    VEMMSG(ubuffer);
```

```
    if (arg->argType != VEM_TEXT_ARG) {
VEMMSG("[args] \"command\" vem-command\n");
vemPrompt();
return(RPC_OK);
    }

    (void) vemCommand(arg->argData.string, spot, cmdList, userOptionWord);
    vemPrompt();

    return(RPC_OK);
}


/*ARGSUSED*/
int
testSelectBySpot(spot, cmdList, userOptionWord)
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
{
    octObject facet, item;
    RPCArg *arg;

    if (lsLength(cmdList) != 0) {
VEMMSG(": select-by-spot\n");
vemPrompt();
return(RPC_OK);
    }

    (void) sprintf(ubuffer, "UOW is %d\n", userOptionWord);
    VEMMSG(ubuffer);

    facet.objectId = spot->facet;

    if (octGetById(&facet) < OCT_OK) {
octError("facet");
return(RPC_OK);
    }

    if (vuFindSpot(spot, &item, OCT_INSTANCE_MASK) == VEM_OK) {
if (set == 0) {
    set = vemNewSelSet(facet.objectId, 0, 65000, 0, 1, 1,
   4, 4, "0001 0010 0100 1000");
}
(void) vemAddSelSet(set, item.objectId);
(void) vemZoomSelSet(set, spot->theWin, 0.3);
    }

    vemPrompt();

    return(RPC_OK);
}
```

```
/*ARGSUSED*/
int
testDeselectByName(spot, cmdList, userOptionWord)
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
{
    octObject facet, item;
    RPCArg *arg;
    Pointer pointer;


    (void) sprintf(ubuffer, "UOW is %d\n", userOptionWord);
    VEMMSG(ubuffer);

    if (lsLength(cmdList) != 1) {
VEMMSG("\"string\" deselect-by-name\n");
vemPrompt();
return(RPC_OK);
    }

    (void) lsFirstItem(cmdList, &pointer, (lsHandle *) 0);
    arg = (RPCArg *) pointer;

    if (arg->argType != VEM_TEXT_ARG) {
VEMMSG("\"string\" deselect-by-name\n");
vemPrompt();
return(RPC_OK);
    }

    facet.objectId = spot->facet;

    if (octGetById(&facet) < OCT_OK) {
octError("facet");
return(RPC_OK);
    }

    item.type = OCT_INSTANCE;
    item.contents.instance.name = arg->argData.string;
    if (octGetByName(&facet, &item) < OCT_OK) {
octError("instance");
return(RPC_OK);
    }

    (void) vemDelSelSet(set, item.objectId);

    vemPrompt();

    return(RPC_OK);
}


/*ARGSUSED*/
```

```
int
testRegionSelect(spot, cmdList, userOptionWord)
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
{
    regObjGen gen;
    octObject facet, object;
    RPCArg *arg;
    Pointer pointer;
    struct octBox box;
    int count = 0;


    (void) sprintf(ubuffer, "UOW is %d\n", userOptionWord);
    VEMMSG(ubuffer);

    if (lsLength(cmdList) != 1) {
VEMMSG("box region-select\n");
vemPrompt();
return(RPC_OK);
    }

    facet.objectId = spot->facet;

    if (octGetById(&facet) < OCT_OK) {
return(RPC_OK);
    }

    (void) lsFirstItem(cmdList, &pointer, (lsHandle *) 0);
    arg = (RPCArg *) pointer;

    if (arg->argType != VEM_BOX_ARG) {
VEMMSG("box region-select\n");
vemPrompt();
return(RPC_OK);
    }

    box.lowerLeft.x = MIN(arg->argData.points[0].thePoint.x,
  arg->argData.points[1].thePoint.x);
    box.lowerLeft.y = MIN(arg->argData.points[0].thePoint.y,
  arg->argData.points[1].thePoint.y);

    box.upperRight.x = MAX(arg->argData.points[0].thePoint.x,
   arg->argData.points[1].thePoint.x);
    box.upperRight.y = MAX(arg->argData.points[0].thePoint.y,
   arg->argData.points[1].thePoint.y);

    (void) fprintf(stderr, "box is (%d,%d), (%d,%d)\n",
   box.lowerLeft.x, box.lowerLeft.y,
   box.upperRight.x, box.upperRight.y);

    if (set == 0) {
set = vemNewSelSet(facet.objectId, 0, 999, 0, 1, 1, 4, 4, "0000 0110 0110 0000");
```

```
    }

    gen = 0;
    if (regObjStart(&facet, &box, OCT_GEO_MASK, &gen, 1) != REG_OK) {
VEMMSG("regObjStart failed\n");
vemPrompt();
return(RPC_OK);
    }

    while (regObjNext(gen, &object) == REG_OK) {
count++;
(void) vemAddSelSet(set, object.objectId);
    }

    (void) regObjFinish(gen);

    if (count > 0) {
(void) vemZoomSelSet(set, spot->theWin, 0.3);
    } else {
VEMMSG("nothing selected\n");
    }

    vemPrompt();

    return(RPC_OK);

}


/*ARGSUSED*/
int
testExit(spot, cmdList, userOptionWord)
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
{
    (void) sprintf(ubuffer, "UOW is %d\n", userOptionWord);
    VEMMSG(ubuffer);

    /* always clear your selected sets before leaving the remote application */
    (void) vemClearSelSet(set);

    /* call RPCExit(0) to exit your application, this does a clean up */
    RPCExit(0);
}



/*
 * dummy functions for passing to Dialog Manager (dm package) functions and RPCUserIO
 *
 * not really need, could use 0 in place of 'dummy/voiddummy', but this
 * gets lint to shut up
```

```
 */
int
dummy()
{
}

void
voiddummy()
{
}


/*
 * this routine handles X events from the X window created in UserMain
 *
 * Note: this is not needed to create an RPC application.
 * This is used to show what can be done in an RPC application.
 */
void
handleXevents(number, read, write, except)
int number;
int *read, *write, *except;
{

    XEvent theEvent;
    long zero = 0;

    printf("handleXevents: number, read, write, except are %d, %d, %d, %d\n",
   number, *read, *write, *except);
    fflush(stdout);

    while (XPending() > 0) {
XNextEvent(&theEvent);
switch (theEvent.type) {

    case ExposeWindow:
XText(theEvent.window, 0, 0, text, strlen(text), theFont,
      BlackPixel, WhitePixel);
XSync();
break;

    case ButtonReleased:
XDestroyWindow(theEvent.window);
XSync();
RPCUserIO(0, (int *) &zero, (int *) &zero, (int *) &zero,
  voiddummy);
break;

    default:
fprintf(stderr, "bad event\n");
break;
}
    }
```

```
    return;
}


RPCFunction CommandArray[] = {
{testOpen, "pane-1", "copy-window", "X", 10},
{testVemMessage, "pane-1", "vem-message", "V", 11},
{testSelectBySpot, "pane-2", "select-by-spot", MYNIL(char), 12},
{testDeselectByName, "pane-2", "deselect-by-name", MYNIL(char), 13},
{testRegionSelect, "pane-2", "region-select", MYNIL(char), 14},
{testVemCommand, "pane-3", "vem-command", MYNIL(char), 15},
{testExit, "pane-3", "exit-testrpc", "E", 16}
};


/*
 * structures for the DIALOG BOXES created in UserMain
 *
 * Note: this is not needed for an RPC Application.  This is used
 * to show what can be done in an RPC Application.
 */
dmWhichItem items[] = {{"button-one", 0, 0},
       {"button-two", 0, 0},
       {"button-three", 0, 0}};

dmWhichItem items2[] = {{"button-one", 0, 0},
        {"button-two", 0, 0},
        {"button-three", 0, 0}};

dmTextItem items3[] = {{"item-1", 1, 20, "<no-value>", 0},
       {"item-2", 2, 10, "initial value", 0}};


#define FDMASK(s) (1 << (s))

/*ARGSUSED*/
int
UserMain(display, spot, cmdList, userOptionWord, array)
char *display;
RPCSpot *spot;
lsList cmdList;
long userOptionWord;
RPCFunction *array[];
{

/* Note: only the 'return(RPC_OK)' is required for UserMain,
 * the rest is used to show what can be done in an RPC application.
 */

    Window theWin;                     /* window openned by testrpc */
    WindowInfo winInfo;
    FontInfo fontStuff;
    int winXCoord, winYCoord;
    int zero = 0;
```

```
    int readmask = 0;
    char buffer[1024];
    char *date;
    int i, selected;
    FILE *logfile;                      /* file to log messages to   */
    octObject facet;                    /* current facet             */
    long tm;                            /* time in seconds           */
    char *name;
    int chrom;


    if ((logfile = RPCfopen("testrpc.log", "a")) == NULL) {
fprintf(stderr, "can not open the log file\n");
RPCExit(-1);
    }

    time(&tm);
    fprintf(logfile, "\n\ntestrpc run at %s\n", ctime(&tm));
    fprintf(logfile, "display is %s\n", display);
    vemGetDisplayType(&name, &chrom);
    fprintf(logfile, "display name is %s, chromatism is %d\n", name, chrom);
    fprintf(logfile, "window is %d, facetId is %d, x,y are (%d, %d)\n",
    spot->theWin, spot->facet,
    spot->thePoint.x, spot->thePoint.y);
    fprintf(logfile, "number of arguments is %d\n", lsLength(cmdList));

    facet.objectId = spot->facet;
    octGetById(&facet);
    fprintf(logfile, "facet is %s:%s:%s (%s)\n\n",
    facet.contents.facet.cell,
    facet.contents.facet.view,
    facet.contents.facet.facet,
    facet.contents.facet.mode);

    fclose(logfile);

    if (lsLength(cmdList) > 0) {
(void) testVemCommand(spot, cmdList, userOptionWord);
    }


    /*
     * the following section creates a X window and sets up input handling
     */

    /* Open the connection */
    if (XOpenDisplay(display) == NULL) {
fprintf(stderr, "Cannot open display '%s'\n", display);
return(RPC_OK);
    }

    /* Get a font */
    font = RPCXGetDefault("vem", "testrpc.font");
    fprintf(stderr, "testrpc.font is %s\n", font);
```

```
    if ((font == MYNIL(char)) || *font == '\0') {
font = "8x13";
    }

    if ((theFont = XGetFont(font)) == NULL) {
fprintf(stderr, "Cannot get font '%s'\n", font);
return(RPC_OK);
    }

    /* Figure out how to center the window */
    XQueryWindow(RootWindow, &winInfo);
    XQueryFont(theFont, &fontStuff);

    /* Figure out how to put the window in the center of the screen */
    winXCoord = winInfo.width/2 - XQueryWidth(text, theFont)/2;
    winYCoord = winInfo.height/2 - fontStuff.height/2;
    theWin =  XCreateWindow(RootWindow, winXCoord, winYCoord,
    XQueryWidth(text, theFont),
    fontStuff.height, 1, BlackPixmap, WhitePixmap);

    XMapWindow(theWin); /* Maps window    */

    readmask = FDMASK(dpyno());

    RPCUserIO(dpyno()+1, &readmask, &zero, &zero, handleXevents);

    XSelectInput(theWin, ButtonReleased|ExposeWindow);
    XFeep(0); XFeep(0); XFeep(0);

    XText(theWin, 0, 0, text, strlen(text), theFont, BlackPixel, WhitePixel);
    XSync();

    /*
     * end of X window section
     */


    (void) dmMultiWhich("RPC DM MultiWhich", sizeof(items) / sizeof(dmWhichItem),
        items, dummy, helpString);

    for (i = 0; i < sizeof(items) / sizeof(dmWhichItem); i++) {
if (items[i].flag == 1) {
    sprintf(buffer, "%s selected\n", items[i].itemName);
    VEMMSG(buffer);
}
    }

    if (dmWhichOne("RPC DM Which One", sizeof(items2)/sizeof(dmWhichItem),
   items2, &selected, dummy, MYNIL(char)) == VEM_OK) {

    sprintf(buffer, "%s selected\n", items2[selected].itemName);
    VEMMSG(buffer);

    } else {
```

```
VEMMSG("nothing selected\n");
    }

    (void) dmMultiText("RPC DM Multi Text", sizeof(items3) / sizeof(dmTextItem),
       items3, MYNIL(char));

    for (i = 0; i < sizeof(items3) / sizeof(dmTextItem); i++) {
sprintf(buffer, "%s is %s\n", items3[i].itemPrompt, items3[i].value);
VEMMSG(buffer);
    }

    (void) vemOpenWindow(&facet, "=100x100+20+60");

    *array = CommandArray;

    VEMMSG("Testrpc initialized\n");
    vemPrompt();

    return(sizeof(CommandArray) / sizeof(RPCFunction));
}
```

```
#! /bin/csh
set path = ($path /usr/new /etc)
echo ""
echo "VEM/RPC DBX"
echo ""
echo "An xterm window will appear, place it"
echo ""
echo "After typing in the name of the remote application to run"
echo "and setting breakpoints, type:"
echo ""
echo alias runrpc \"run $*\" >> /tmp/dbxinit.$$
echo            runrpc
echo ""
xterm $1 -n "VEM/RPC DBX" -e dbx -c /tmp/dbxinit.$$
/bin/rm /tmp/dbxinit.$$
exit
```

5.8 RPC Protocol Interface

This chapter is under major revision at this time. Hopefully this along with the introductory material will give you an idea of how the low level RPC stuff works.

5.8.1 Starting the Remote Process

Using the ULTRIX *inet* or *decnet* protcols, a remote process is started by the `rsh` command. The command is created using the following:

```
sprintf(command, "rsh %s %s -n %s %s %s %d %s %d %d %d %d %d",
                 remotehost, remoteuser,
                 path, display, localhost, port, protocol,
 byte1, byte2, byte3, byte4, integer);
```

5.8.2 Rendezvous with VEM

The first act of the remote application is to connect back to the VEM server on the host and port passed to the remote application.

5.8.3 Initialization

Once the connection is made, the remote application determines the byte ordering by combining `byte1` through `byte4` and comparing it with `integer`. The server does no byte swapping, all byte swapping is handled by the application. Once this is done, the application calls the server function `VEM_INITIALIZE_APPLICATION` with the arguments being the version of the RPC library, and the menu/pane entries for the remote application. The menu/pane entries consist of a `long` count of the number of entries following by that number of string pairs, the first being the pane name and the second being the menu item name. At this time a floating point number is passed so the server and application can determin if they have the same floating point formats.

5.8.4 Packets to VEM

A packet from the the remote application to VEM consists of a function number followed by the arguments to the function. The function number is a `long` and the arguments are sent as described in Section 5.8.5. If the function does not return any data, the remote application does not wait for a return code and continues on. If the function does return data, the remote application waits for a return code and the data.

5.8.5 Sending Data

All data is sent in VAX byte order (not NETWORK byte order). `long, short` and pointer (non-string pointer) data are passed as longs. Strings are passed as a `long` count (string length) and a sequence of bytes (not including the trailing zero byte). `float` and `double` data items are passed as 8 bytes if the floating point formats are the same and as the `sprintf %g` string representation if they are different. `NULL` string pointers are sent as a zero count. The basic data item sent between the RPC application and VEM is the `octObject.` All data with structure (such as the `octObject`) is sent in a top down, left to right recursive manner. To make this clearer, the C `struct` definitions in Figure 5.1 describe an `octObject`, specifically an `OCT_INSTANCE`. RPC sends an `OCT_INSTANCE` as shown in Figure 5.2.

```
struct octPoint {
    octCoord x;
    octCoord y;
};
struct octTransform {
    struct octPoint translation;
    octTransformType transformType;
    double generalTransform[2][2];
};
struct octInstance {
    struct octTransform transform;
    char *name;
    char *master;
    char *view;
    char *version;
};
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 octCircle circle;
        struct octPath path;
        struct octLabel label;
        struct octBag bag;
        struct octLayer layer;
        struct octPoint point;
        struct octEdge edge;
    } contents;
};
```

Figure 5.1: C Language Structure Definitions for an `OCT_INSTANCE`

- Send `type`
- Send `objectId`. Some OCT functions only use this field and in those cases, only this field is sent.
- Send `contents`
  - Send `octTransform`
    - Send `translation`
      - Send `x`
      - Send `y`
    - Send `transformType` (an integer)
    - Send `generalTransform`. This is only sent for `OCT_FULL_TRANSFORM`.
  - Send `name`
  - Send `master`
  - Send `view`
  - Send `version`

Figure 5.2: Sending an `OCT_INSTANCE`

Chapter 6

Customizing VEM

David S. Harrison
University of California, Berkeley

6.1 Introduction

VEM is an interactive graphics shell/editor for IC designs represented using the OCT CAD/VLSI data representation system. The primary purpose of VEM is to provide a means of looking at the graphics representation of OCT views and invoking various CAD tools on these views. VEM also provides standard graphics editing capabilities for physical, symbolic, and schematic design.

VEM is graphics tool which runs on both color and black and white workstations which support the X window system. Since the number of possible choices of colors, fill patterns, font styles, and other parameters is quite large, VEM provides a means for a user to change most of these parameters on a user by user basis.

VEM reads two text files and one OCT facet to determine the values of various parameters. The most important of these is the *X defaults* file (`~/.Xdefaults`). This text file contains many of the color and font style parameters used by VEM. It is described in the next section. The second file (also text) is the *VEM bindings file* (`$OCTTOOLS/lib/vem/vem.bindings.VERSION`). This file contains the keyboard and menu bindings for all VEM commands. It is described in section 6.3. Finally, VEM uses the *Technology Access Package* (tap) to determine the display characteristics of geometry attached to certain layers. TAP stores these display characteristics in an OCT facet (`$OCTTOOLS/tech/<tech-name>/tap.views`). Changing the display attributes of layers is discussed in the last section.

6.2 VEM Default Parameters

Like other X application programs, VEM reads a file called `.Xdefaults` in a user's home directory for most of its user preference parameters. The format of the file consists of lines of the following form:

```
application.parameter: value
```

Normally, all VEM parameters are prefixed by "vem". However, the program looks for parameters based on the invocation name of the program (`argv[0]`). Thus, other sets of defaults can be defined by creating symbolic links to the program with other names.

There are eight value types VEM accepts as parameter values. These are integers, strings, real numbers, colors, fonts, line styles, fill patterns, and attributes. The legal values for all these types (except attributes which are internal) are given below:

Integers, Reals, and Strings Integer, Real, and String values are specified in the normal fashion. Integers must be a string of decimal digits possibly prefixed with a minus sign. Real numbers are specified in `%f` format (see *scanf(3S)*). Strings are specified by a string of characters terminated with a newline. No quotes are required.

Colors A color may be the name of a color as it appears in `/usr/lib/rgb.txt` or a direct specification. The format of a direct specification is a hexadecimal constant of the form: `#RGB`, `#RRGGBB`, or `#RRRRGGGGBBBB`, where R, G, and B are relative intensities of red, green, or blue respectively. The three forms simply specify the relative intensities with varying resolution.

Fonts The font name of any legal X font may be used as a value to a font parameter. The directory containing X font names is normally `/usr/lib/Xfont`. The suffix `.onx` of font files found in this directory should be omitted when specifying the font name in `~/.Xdefaults`.

Line Styles Line styles are specified as a string of ones and zeros less than 16 characters in length. The line style pattern is constructed by repeating the supplied pattern a sufficient number of times to construct a 16 character bit pattern. This pattern is used by X to draw the line.

Fill Patterns Fill patterns specify a stipple used for drawing filled geometry. The format of a fill pattern specification is: `<rows>,<cols>,<one-zero-str>`. `<rows>` and `<cols>` specify the size of the pattern and `one-zero-str` is a string of ones and zeros specifying the pattern. The following example specifies a right cross-hash pattern:

```
4,4,0001001001001000
```

The following subsections contain descriptions of all VEM X defaults grouped by subject. Each subsection contains a description of the class of defaults described and a list of default entries. A default entry has the following form:

Parameter Name – *type*: "b/w", "color". Parameter description.

Parameter Name specifies the name of the X default as it would normally appear in the X default file. *type* specifies the type of the parameter (one of *Integer, Real, String, Color, Line Style* or *Fill Pattern*). Following the type is the default value for the parameter enclosed in double quotes. Some parameters have different default values based on whether the display is color or black and white. If this is the case, the black and white default precedes the color default.

6.2.1 General Purpose

General purpose parameters are those which don't fit into any other category. These parameters are described below.

vem.logfile – *String.* VEM writes to a log file all error and status messages. This specifies the name and location of a file where this information will be written. By default, these files are deposited in `/tmp`. The name of the log file is of the form `vem.log.DDDDDD`, where `DDDDDD` is replaced with a unique number. It is recommended that users change this default to one of the form `/tmp/vem.log.USER`, where `USER` is replaced with the user name. This makes it easier to find the log file if there are problems. **Note:** if VEM cannot write to the specified file, log messages will be written to standard output.

vem.version – *String.* This is default version used when opening new cells. It defaults to the null string.

6.2.2 Overall Display

On color displays, VEM tries to allocate colors in a manner that allows it to show argument entry and selected information on its own plane. This makes both argument entry and selection much faster. However, to do this VEM must allocate a large number of colors which cannot be shared with other applications. Furthermore, the number of colors is fixed for the session and cannot increase.

VEM allows the user to specify two parameters which control the allocation of colors: *vem.colors* and *vem.planes*. When VEM begins execution, it will attempt to allocate $n2^p$ color cells where n is *vem.colors* and p is *vem.planes*. If successful, this allows VEM to display p independent selected sets on their own plane with n colors used for layer display. Note that VEM displays instances with colors which are slightly different from normal layer colors. Thus, if a technology has l layers, VEM will need about $2l + 5$ colors to display it (there are five other colors VEM also uses in graphics windows). Once VEM runs out of allocated colors, it will attempt to reclaim the "alternate" colors used for instance display and continue. If VEM runs out of allocated colors and all alternate colors have been reclaimed, VEM will use black to display the new items. If VEM cannot initially allocate the requested number of planes and colors, VEM will not allocate any planes for argument entry or selection and use shared colors for layer display. The color allocation parameters are described below:

vem.colors – *Integer:* "25". Specifies the number of colors VEM is allowed to consume in all of its graphics windows. This parameter has no meaning on black and white displays.

vem.planes – *Integer*: "1". This specifies the number of planes VEM can use for displaying selected information. This default is only meaningful on color displays with writable color maps. Normally, VEM uses one plane to show all standard selected information. If you specify more than one plane, these may be allocated by RPC applications for their use. However, this resource is scarce. On eight-plane machines, a maximum of three planes can be used (assuming the default number of *vem.colors*) before exhausting the color resources of the machine.

The remaining parameters described in this section specify drawing parameters and the overall appearance of VEM graphics windows.

vem.abstraction – *Integer:* "0". This specifies the maximum size of a contact (in pixels) in which an abstraction of the contact should be drawn. This can be used to tell VEM to draw contacts quickly (not treating them as first class instances). VEM determines whether an instance is a contact by checking to see if it is connected to the bag called "CONNECTORS". Note that by default, this option is off (its value is zero). When viewing large layouts, a good value for this parameter is 20.

vem.solid.threshold – *Integer:* "12". The size (in pixels) before a shape is drawn using solid fill rather than stipple fill. A large number specifies all geometry should be drawn solid regardless of the current scale factor.

vem.background.color – *Color:* "white", "LightGray". This is the background color for all VEM graphics windows.

vem.border.color – *Color:* "black", "white". This is the border color of all VEM graphics windows.

vem.border.width – *Integer:* "2". This parameter specifies the border width of VEM graphics windows in pixels.

vem.cursor.color – *Color:* "black", "white". This is the color of the mouse cursor when inside VEM graphics windows.

vem.nolayer.color – *Color:* "black". If VEM cannot find the display attributes for a layer in the technology for a facet, it will display the layer using this color.

vem.nolayer.fill – *Fill Pattern:* "1,1,1". Specifies the fill pattern to be used when displaying layers which have no display attributes defined in the technology for the cell.

vem.nolayer.style – *Line Style:* "1". This specifies the outline style used for outlining geometries on layers which are not in the technology for a facet.

6.2.3 Argument Entry

Argument entry refers to the mechanism used by VEM to enter boxes, lines, and points onto the VEM argument list. The parameters below allow the user to exert control over this process.

vem.snap – *Integer:* default value same as *vem.lambda*. All graphics input is snapped to this number of Oct units.

vem.argument.pixels – *Integer:* "20". Number of pixels the mouse must move before a point becomes a box or line.

vem.lambda – *Integer:* "20". This is the number of Oct units per lambda. Coordinates returned by the *where* command and those shown in the upper right corner of the title bar are computed using this conversion factor.

vem.gravity – *Integer:* "10". When editing large schematic diagrams, it can be difficult to accurately draw paths which intersect actual terminals or other paths. Thus, VEM attempts to ease this task by snapping the endpoints of line arguments to nearby terminals or paths automatically. This parameter tells VEM how far to look (in pixels) for nearby terminals or or paths.

vem.select.color – *Color:* "black", "cyan". This color is used to display graphics argument types and to highlight selected geometries.

vem.select.style – *Line Style:* "`00001111`", "1". This is the line style used to highlight selected geometries.

vem.select.fill – *Fill Pattern:* "4,4,0000011001100000". Specifies the fill pattern used to highlight objects which have been selected. Only the black and white default is given above. The color default is similar but uses a less dense pattern of dots. Other argument related options (such as manhattan line snapping) can be set on a window by window basis using the *flag-options* command.

6.2.4 Window Options

VEM allows the user to specify on a window-by-window basis a large number of options which control the appearance of graphics windows. During execution, these parameters can be changed using the *flag-options* command. The window options parameter described below can be used to specify the default values for newly opened windows.

vem.window.options – *String:* "gtedabcrf-imG". When a new window is opened, VEM uses this string parameter to determine the default window characteristics. It consists of single character options which may appear before or after a minus sign. Those appearing before the minus sign are considered "on" and those appearing after are considered "off". The character options are described below:

g Display a grid in the window.

t Display a title bar in the window.

e Show instances expanded (*contents* facets).

d Show argument size coordinates in title bar.

a Show absolute cursor location in title bar.

b Show status bits in title bar.

f Show edit cell in title bar.

c Show context cell in title bar.

r Show RPC application name in title bar.

i Show instance bounding boxes in window.

m Snap line arguments to manhattan angles.

A Show actual terminal frames and names.

G Turn on line endpoint gravity.

Notes: If specified, *all* letters must appear in the parameter specification. The manhattan lines option is automatically turned on when editing symbolic or schematic cells. The gravity option is automatically turned on when editing schematic cells.

6.2.5 Grid Display

Normally, VEM displays a grid in all graphics windows. This grid consists of three components: zero axes, a minor grid, and a major grid. The zero axes are lines corresponding to the X and Y axes and are always drawn (they cannot be turned off). The minor grid consists of lines or dots usually drawn at one lambda spacing. The major grid also consists of lines or dots but is usually drawn at a larger spacing than the minor grid for reference purposes. The basic grid parameters are described below:

vem.grid.major.units – *Integer:* "100" (if logarithmic, "10"). This integer parameter describes the number of OCT units per major grid line. If a logarithmic grid is in effect, it specifies the constant which is multiplied times the nearest integral power of the base to determine the grid size. See *vem.grid.base* for details.

vem.grid.minor.units – *Integer:* "20" (if logarithmic, "2"). Normally, this specifies the number of OCT units per minor grid line. If logarithmic grid lines have been selected using *vem.grid.base*, this specifies the multiplicative constant for minor grid lines.

vem.grid.pixels – *Integer:* "8". If the number of pixels between major or minor grid lines is less than this parameter, the grid lines will not be drawn.

vem.grid.minor.size – *Integer:* "1". This parameter specifies the width of the minor grid lines in pixels.

vem.grid.major.size – *Integer:* "1". This parameter specifies the size of the major grid lines or dots in pixels.

vem.grid.zero.color – *Color:* "black", "white". This is the color used when drawing the zero grid lines (the axes).

vem.grid.zero.size – *Integer:* "3", "1". This parameter specifies the width of the zero grid lines in pixels.

vem.grid.zero.style – *Line Style:* "1". This specifies the line style of zero grid lines.

Grids can be drawn as a series of lines or dots. The following parameter determines the grid style:

vem.grid.dots *Binary:* "1", "0". If set, the grid displayed in graphics windows will be dotted. If not set, the grid will be displayed as a series of lines. Dotted grids are most effective on black and white displays and when editing schematic diagrams (VEM automatically turns on dotted grids under both of these conditions). Line grids are most effective when displaying layout.

If grids are drawn as lines, the following parameters determine the Appearance of those lines:

vem.grid.major.line.color – *Color:* "black". Specifies the color of major grid lines.

vem.grid.major.line.style – *Line Style:* "1". This is the line style used for drawing major grid lines.

vem.grid.minor.line.color – *Color:* "black", "#888". This is the color used to draw minor grid lines.

vem.grid.minor.line.style – *Line Style:* "01", "1". This is the line style pattern used to draw minor grid lines.

If grids are drawn as dots, the following parameters determine the appearance of those dots:

vem.grid.major.dot.color – *Color:* "black", "white". This specifies the color of grid dots at major grid spacings.

vem.grid.minor.dot.color – *Color:* "black". Specifies the color of grid dots at minor grid spacings.

If the scale factor of a window is such that major or minor grid would be too fine to display, VEM normally does not draw the grid. However, it is possible to specify that VEM draw grids *logarithmically*. If this option is turned on, VEM will choose a power of the basic grid spacing and always draw a grid regardless of the scale of the window. The following parameters can be used to control this feature:

vem.grid.base – *Integer:* "0". If non-zero, this option selects a logarithmic grid. Normally, there are two grids drawn at a fixed number of Oct units (major and minor grid lines). In a logarithmic grid, grids are drawn at some constant (specified by the major.units and minor.units values) times the nearest integral power of the grid base. For example, if the constant is two and the base is 10, grids will be drawn at 2, 20, 200, etc. Note that this option is normally off (its value is zero).

vem.grid.minbase – *Integer:* "1". This option specifies the smallest base to be used for drawing logarithmic grids (see *vem.grid.base* for details about logarithmic grids).

6.2.6 Instance Display

In the current version of VEM the instance hierarchy is drawn in a depth-first manner (*i.e.* instances are drawn as objects). Furthermore, at each level, VEM can choose to display either the *contents* facet or an *interface* facet. By default, VEM does not show the name or master of an instance when drawing an instance. All of these features can be controlled using the following parameters:

vem.inst.priority – *Integer:* "1". This parameter specifies the relative drawing priority of instances. Grid lines are drawn at a priority level of zero. Higher priorities are drawn last.

vem.interface – *String:* "interface". This string parameter specifies the name of the interface facet. This facet will be used to draw instances in unexpanded mode.

vem.inst.color – *Color:* "black", "white". This specifies the color used when drawing an instance bounding box.

vem.inst.style – *Line Style:* "1". This is the line style used when drawing an instance bounding box. The default is 1 (solid).

vem.boundbox.min – *Integer:* "8". This integer parameter specifies the maximum size of a terminal (in pixels) where it is acceptable to draw an abstraction of the terminal. Normally, a terminal is drawn as a highlighted outline with the terminal name inside this outline. However, if the terminal is smaller than this parameter, it is simply drawn as a filled box in the appropriate color.

vem.boundbox.font – *Font:* "6x10". Bounding boxes are drawn for unexpanded instances without interfaces, formal terminals, and actual terminals. The instance or terminal name is drawn inside this bounding box. This parameter specifies the font used to draw the instance or terminal name.

vem.boundbox.thres – *Real:* "2.0". Labels drawn in bounding boxes may or may not be drawn depending on the size of the bounding box. This parameter specifies how many times wider the text may be than the box before the label is not drawn.

vem.boundbox.color – *Color:* "white". Instance bounding boxes are drawn using this color.

6.2.7 Terminal Display

Terminals are used to show legal connection points between levels of the design hierarchy. At any given level, VEM displays two kinds of terminals: formal terminals and actual terminals. Formal terminals are those which specify the interface of the current cell. Actual terminals are the legal connection points of the instances found in the current cell. The display of these terminals can be controlled by the following parameters:

vem.formal.color – *Color:* "black", "white". This is the color used to draw the outline around formal terminals.

vem.actual.color – *Color:* "black", "`#00FFFF`". This is the outline color of actual terminals.

vem.actual.style – *Line Style:* "`11111100`", "`01`". This is the line style used to outline actual terminals.

vem.formal.style – *Line Style:* "0111", "1". This is the line style used when drawing the outline around formal terminals.

vem.boundbox.min – *Integer:* "8". This integer parameter specifies the maximum size of a terminal (in pixels) where it is acceptable to draw an abstraction of the terminal. Normally, a terminal is drawn as a highlighted outline with the terminal name inside this outline. However, if the terminal is smaller than this parameter, it is simply drawn as a filled box in the appropriate color.

vem.boundbox.font – *Font:* "6x10". Bounding boxes are drawn for unexpanded instances without interfaces, formal terminals, and actual terminals. The instance or terminal name is drawn inside this bounding box. This parameter specifies the font used to draw the instance or terminal name.

vem.boundbox.thres – *Real:* "2.0". Labels drawn in bounding boxes may or may not be drawn depending on the size of the bounding box. This parameter specifies how many times wider the text may be than the box before the label is not drawn.

6.2.8 Console Window Appearance

The console window is used by VEM to display error and help messages and to echo user input. The window can display up to three different fonts each in its own color. The *normal* font is used to display VEM messages. The *bold* font is used to echo user input. The *italic* font is used by remote applications to distinguish their output from that of VEM. The attributes of all console related features can be controlled by specifying the following parameters:

vem.console.bgnd – *Color:* "white", "`#EEE`". This parameter specifies the background color of the console window.

vem.console.fgnd – *Color:* "black". This is the color used to draw the border around the console window.

vem.console.cursor – *Color:* "black", "red". This specifies the color of the mouse cursor when it is inside the console window.

vem.text.color.norm – *Color:* "black", "VioletRed". This specifies the color of the standard VEM normal font. For example, this font is used to display VEM messages in the console window.

vem.text.font.norm – *Font:* "vtsingle", "helv12". This is the X font used for the standard VEM normal font. *Note:* On black and white machines, fixed width fonts seem to work better due to an unknown X bug. Thus, fixed width fonts have been chosen for black and white displays.

vem.text.color.bold – *Color:* "black". This specifies the color of the standard VEM bold font. For example, this font is used to display user input in the console window and in type-in fields of dialog boxes.

vem.text.font.bold – *Font:* "vtbold", "helv12b". This is the X font used for the standard VEM bold font.

vem.text.color.italic – *Color:* "black", "blue". This specifies the color of the standard VEM italic font. For example, this font is used in dialog buttons and blender controls for example.

vem.text.font.italic – *Font:* "9x15", "helv12i". This is the X font used for the standard VEM italic font.

vem.JumpScroll – *Binary:* "0". If on, the line-to-top and top-to-here operations in the console window will not scroll to the target position smoothly. Instead, the window will be repainted once at the correct spot.

6.2.9 Menu Appearance

VEM uses a two level deck-of-cards menu system developed by Project Athena at MIT. This is an independent package which reads its own set of parameters (but with the VEM prefix). These parameters are described below:

vem.MenuFreeze – *Binary:* "0". If non-zero, the menu package grabs the server while posting a menu and attempts to save the information under the menu. This makes menus much faster. Its use is recommended.

vem.MenuBackground – *Color:* "white". Determines the menu background color.

vem.MenuMode – *String:* "invert". Specifies the highlight mode of the menu. One of *box* or *invert*. Box draws a box around the item, invert inverts the selection.

vem.MenuMouse – *Color:* "black". Specifies the color of the mouse when in the menu.

vem.MenuStyle – *String:* "right_hand". Customization for right or left handed people. One of *left_hand*, *right_hand*, or *center*.

vem.PaneStyle – *String:* "center". Determines the display style of menu panes. One of *flush_left*, *flush_right*, or *center*. The default is *center*.

vem.SelectionStyle – *String:* "flush_left". Justification of menu selections. One of *flush_left*, *flush_right*, or *center*.

vem.PaneFont – *Font:* "9x15". Font used for pane labels.

vem.SelectionFont – *Font:* "6x10". Font used for menu selections.

vem.PaneForeground – *Color:* "black". Foreground color for the pane label.

vem.SelectionForeground – *Color:* "black". Color of menu selections.

vem.PaneBorder – *Color:* "black". Color of pane borders.

vem.SelectionBorder – *Color:* "black". Color of menu selection borders.

vem.PaneBorderWidth – *Integer:* "2". Width of pane borders.

vem.SelectionBorderWidth – *Integer:* "1". Width of menu selection borders.

vem.PaneSpread – *Real:* "1.0" Determines horizontal spread of menu panes. Specifies ratio of horizontal to vertical spread.

vem.SelectionSpread – *Real:* "0.25". Determines the inter-selection padding. A value of 1.0 specifies a padding the height of the selection font.

vem.MenuInactivePattern – *String:* "gray3". Determines which of five patterns is used for inactive panes. One of *dimple1*, *dimple3*, *gray1*, *gray3*, or *cross_weave*.

6.2.10 Dialog Box Appearance

VEM uses a forms-based input package called XDM to present dialogs to the user. Inside these dialogs, VEM tries to enforce the following display conventions: dialog titles and type-in fields are displayed in the standard VEM bold font, button labels are displayed in the standard VEM italic font, blender control and type-in labels are displayed in the standard VEM normal font. The values for these and other dialog related parameters can be changed by specifying the parameters below:

vem.DialogBackground – *Color:* "white". Specifies the background color of dialog boxes.

vem.DialogBorderColor – *Color:* "black". Border color of dialogs.

vem.DialogBorderWidth – *Integer:* "2". Border width of dialogs.

vem.DialogCursorColor – *Color:* "black". Mouse cursor color when cursor is in dialog.

vem.dialog.freeze – *Binary:* "1". This parameter is used primarily for debugging. If the value is zero, all pop-up dialogs will not freeze the X server. This facilitates the use of source-level debuggers while pop-up dialogs are posted. The default value is one (pop-up dialogs freeze the server).

vem.text.font.norm – *Font:* "vtsingle", "helv12". This is the X font used for the standard VEM normal font. *Note:* On black and white machines, fixed width fonts seem to work better due to an unknown X bug. Thus, fixed width fonts have been chosen for black and white displays.

vem.text.color.norm – *Color:* "black", "VioletRed". This specifies the color of the standard VEM normal font. For example, this font is used to display VEM messages in the console window.

vem.text.font.bold – *Font:* "vtbold", "helv12b". This is the X font used for the standard VEM bold font.

vem.text.color.bold – *Color:* "black". This specifies the color of the standard VEM bold font. For example, this font is used to display user input in the console window and in type-in fields of dialog boxes.

vem.text.font.italic – *Font:* "9x15", "helv12i". This is the X font used for the standard VEM italic font.

vem.text.color.italic – *Color:* "black", "blue". This specifies the color of the standard VEM italic font. For example, this font is used in dialog buttons and blender controls for example.

6.2.11 Title Bar Display

By default, VEM displays a title bar above each graphics window. On the left side of the bar, VEM tracks the size of the current graphics argument (box, line, etc). On the right side of the bar, VEM tracks the current cursor position. In the center of the title bar, VEM displays the name of the cell, status bits, and the name of the RPC application (if any). The basic Appearance of the title bar can be controlled using the following parameters:

vem.title.backgr – *Color:* "black", "`#602`". This is the background color of the title bar displayed above VEM graphics windows.

vem.title.color – *Color:* "white", "goldenrod". This is the color used to draw items in the title bar displayed above VEM graphics windows.

vem.title.font – *Font:* "6x10", "timrom10". This is the font used to draw window titles.

Controlling the items displayed in the title bar can be done using the *vem.window.options* parameter.

6.2.12 Schematic Entry Parameters

Schematic entry in VEM is an extension of symbolic where the primitives are schematic symbols and path width and layer are insignificant. The parameters below describe some of the schematic specific options VEM supports:

vem.schem.layer – *String:* "WIRING". This default specifies the layer to use for creating wires in a schematic diagram.

vem.gravity – *Integer:* "10". When editing large schematic diagrams, it can be difficult to accurately draw paths which intersect actual terminals or other paths. Thus, VEM attempts to ease this task by snapping the endpoints of line arguments to nearby terminals or paths automatically. This parameter tells VEM how far to look (in pixels) for nearby terminals or or paths.

6.2.13 Sample X Defaults

All of the defaults described in previous sections are optional. VEM defines defaults for all of them. However, as an example, below are the parameters the author defines in his `~/.Xdefaults` file:

```
vem.MenuFreeze: on
vem.MenuMouse: red
vem.MenuBackground: DarkSlateGray
vem.MenuFont: timrom12b
vem.PaneForeground: white
vem.PaneFont: helv12b
vem.SelectionForeground:yellow
vem.SelectionBorderWidth:2
vem.SelectionSpread: 0.0
vem.SelectionStyle: center
vem.SelectionFont: timrom10

vem.DialogFont: helv12b
vem.DialogBackground: #cce
vem.DialogForeground: Black
vem.DialogCursorColor: red
vem.dialog.freeze: 1

vem.abstraction; 20
```

```
vem.menu.delay: 20
vem.log: /tmp/vem.log.davidh
vem.grid.base: 10
vem.border.color: wheat
vem.boundbox.thres: 1.0
```

6.3 VEM Key, Menu, and Command Bindings

The bindings file describes the mapping between a command name and its key and menu bindings. By default, VEM reads a bindings file which is provided with the VEM release. A user defines his own defaults by creating his own bindings file which includes the standard bindings, adding any overriding bindings to the end of the file, and modifying the *vem.bindings* parameter described below to point at the new bindings file.

vem.bindings – *String:* "`$OCTTOOLS/lib/vem/vem.bindings.<version>`". This is a string specifying the location of the standard VEM bindings file. Note that the actual name of the standard bindings file depends on the version of VEM (e.g. `~octtools/lib/vem/vem.bindings.6-0`).

Each line of the bindings file is a statement. There are four kinds of statements: normal binding, remote declaration, copy directive, and include directive. A normal binding sets up key, menu, and alias bindings for a specified command. A remote declaration defines a new command which starts a remote application. The copy directive copies a previously defined table into a new table. The include directive reads bindings from some other file. A special form of the include directive is available for automatically including the standard VEM bindings. The format for each statement type is described below. Capitals indicate keywords.

viewtype id [KEY keyspec ...] [MENU menuSpec ...] [ALIAS aliasSpec ...] This is the format for a standard binding. The viewtype and unique command name (id) are required. The KEY, MENU, and ALIAS clauses may appear in any order and any number of times. A keyspec must be a literal character, hat notation, or octal notation. A menuspec consists of a pane name (specifying the pane of the entry) and a selection index (an integer indicating how far down in the pane the entry will appear). An alias spec is simply another name for the command. If ALIAS appears without any specifications, the standard display name for the command is registered as an alias. The keywords may be plural (e.g. KEYS, MENUS, or ALIASES).

REMOTE id host path This statement is not a binding but a declaration specifying *id* as a remote application. The host and path specify the location of the program executable. Once an id has been declared as remote, bindings to it may be made using the other statement types. However, the declaration must precede the use of the id in a standard binding.

new-viewtype COPY old-viewtype This is a directive which tells VEM to create a new binding table which is an exact copy of the old table *up to the definitions specified so far.*

INCLUDE file-name This directive reads bindings from the file named *file-name*. It is most often used to include the current standard bindings at the top of a user defined bindings file. The file specification may include tilde notation.

INCLUDESTD This directive is like the *include* directive except it automatically includes the standard VEM bindings file (`$OCTTOOLS/lib/vem/vem.bindings.<version>`).

Comment lines (lines prefixed with a pound-sign (`#`)) are ignored.

6.4 Technology Information

This version of VEM uses the Technology Access Package (*tap*) to determine the display attributes of layers, minimum path width sizes, and primitive palettes (see the *palette* command). TAP stores this information in an OCT cell in a standard location. The following steps are used by TAP to find the technology facet for a given cell:

1. First, the value of the TAP root directory is consulted to determine the root where all technologies are found. By default, this directory is `$OCTTOOLS/tech`. However, VEM allows you to specify this location using the following parameter:

   vem.technology – *String:* "`$OCTTOOLS/tech`". This default specifies the root directory used by VEM to find technology information.

2. The value of the string property `TECHNOLOGY` attached to the cell is appended onto the root directory given above.

3. The cell name `tap.views` is appended onto the results of the previous step. The path now specifies the location of the technology cell.

4. The view of the technology is determined by examining the `VIEWTYPE` property attached to the cell. If there is no `VIEWTYPE` property, the view name of the cell is used instead.

For example, if a cell has a `TECHNOLOGY` property of "scmos" and a `VIEWTYPE` property of "schematic", TAP will attempt to open the cell `$OCTTOOLS/tech/scmos/tap.views:schematic`.

When VEM opens a new cell, it prompts the user for the values of the `TECHNOLOGY`, `VIEWTYPE`, and `EDITSTYLE` properties. The default value for the `TECHNOLOGY` field can be set by the following parameter:

vem.deftechname – *String:* "scmos". This parameter specifies the default value used when VEM prompts for the `TECHNOLOGY` property of a new cell.

The OCT cell used to store technology information (called the technology facet) contains information about the display of layers (colors, fill patterns, border styles, etc.), design rule information, and instances of cells that contain common primitives (palettes). The display information can be customized on a device-by-device basis. There are two ways to change the technology information in a TAP view: *tech-edit* and *pat2tap*. Tech-edit is an RPC application which runs under VEM and allows you to change the current technology information interactively using a dialog-based interface. Pat2tap is a program which converts an existing `vem.patterns` file into a form TAP understands (See Chapter 7).

Chapter 7

Creating a New Technology

Describing a new technology is no easy task, let that be said right away. You need a very good understanding of the technology you want to describe, and some familiarity with the **Octtools 5.2**. This chapter tells you about the tools that will help you build the technology description.

Technologies are described by **tap** views which are OCT facets. **tap** views can be created and modified by **tech-edit** and **techBuild** (formerly called **pat2tap**) and checked with **tap-print**. Tap views can be programmatically accessed by the **tap** programming library and by the helper package **th**.

Technologies are divided into views, where each view corresponds to an OCT view type. Currently, there are `PHYSICAL`, `SYMBOLIC`, and `SCHEMATIC` view types. The `PHYSICAL` view type corresponds to mask-level geometries and is used for entering leaf cells. The `SYMBOLIC` view type corresponds to a more abstract level that deals mainly with the interconnectivity of cells. The `SCHEMATIC` view type is yet a further abstraction that deals mainly with the abstract interconnectivity of symbols.

Each view of the technology is divided into four parts:

1. display attributes; The display attributes describe how to display each of the layers in the technology (color, fill patterns, border patterns, drawing priority) for each type of display device.
2. design rules; The design rules describe a basic set of design rules: minimum width, same layer spacing, different layer spacing, and overlap.
3. electrical parameters; The electrical parameters include for example sheet resistance and capacitance to substrate of each layer.
4. palettes. The palettes are used by several tools to get information on available layers and cell libraries, but they are most useful inside **vem**. There are two important palettes: the layer palette and the connector palette. The layer palette describes the layers that exist in the technology. The connector palette describes the vias and bends that are available to the tools. The connector palette only exists for `SYMBOLIC` and `SCHEMATIC` view types.

7.1 Display Attributes

The first step in creating a technology description for a view type is to describe the display attributes for the layers. You should use the program **techBuild** to create the layers. You must first create an ASCII file that contains descriptions of the layers. This file is commonly known as the `vem.patterns` file, but any file name will do. In the scmos technology, the pattern files are in the `$OCTTOOLS/tech/scmos/patterns` directory – use those as complete examples. The pattern file contains a lisp-like description of each layer. The description of the `PHYSICAL` and `SYMBOLIC` views type will have all layers[1].

[1] The `SYMBOLIC` view should only have the interconnect layers, but due to the way interface facets are represented, all layers must be in the `SYMBOLIC` view.

In **Octtools 5.2** there is also a *schematic* technology, which allows only views of type `SCHEMATIC`. This technology has two fundamental layers, `WIRING` and `SYMBOL`, and a few other layers `RED, GREEN, BLUE` to provide for colored labels.

The syntax of the pattern file is easy, even if the keywords are, for historical reasons, sort of bizarre. For example, the description of each layer begins with the keyword `figureGroupDefault`, and the keyword `technology` really means "collection of display attributes." The syntax also allows you to specify the minimum width for each layer, but you should not do it, because there is a better method (see next section).

Here is a description of the syntax

```
(technology viewName
   (comment "run techBuild on vem.pattern files to create tap views")
   (comment "viewName is normally the VIEWTYPE, but could be anything")
   (figureGroupDefault layerName  (comment "name of the layer, i.e. IMPL, NDIF")
        (userData priority -10)   (comment "drawing priority, lowest first")
        (color 0 1000 0)          (comment "rgb values, range is 0-1000")
        (width 20)                (comment "minimum width for the layer")
        (fillPattern 4 4
                "0111
                 1111
                 1111
                 0111")           (comment "rows cols bitPattern")
                                  (comment "solid fill is 1 1 '1'")
   )
   (comment "one figureGroupDefault for each layer in the VIEWTYPE")

   (comment "do not use width, even if it is allowed by the language" )
   (comment "use tech-edit instead to set the minimum width of each layer" )

)
```

Here is a fragment of a `vem.patterns` file used to create the layers for the `PHYSICAL` view of the *scmos* technology.

```
(technology physical-color
   (comment "PHYSICAL view of the SCMOS technology for color displays")

   (figureGroupDefault NDIF
        (comment "N-type diffusion")
        (userData priority -10)
        (color 0 1000 0)               (comment "green")
        (fillPattern 1 1 "1"))

   (figureGroupDefault PDIF
        (comment "P-type diffusion")
        (userData priority -5)
        (color 698 322 243)        (comment "brown")
        (fillPattern 1 1 "1"))

   (figureGroupDefault POLY
        (comment "Polysilicon")
        (userData priority 10)
        (color 1000 100 100)        (comment "red")
        (fillPattern 4 4
                "0111
                 1111
                 1111
                 0111"))

   <<<< more of the same stuff deleted >>>

)
```

The view type name in the `vem.patterns` file is just a comment, the actual view type name is entered when you run **techBuild**. Your `vem.patterns` file can contain multiple view type descriptions and multiple display descriptions.

tap supports different colors and fill patterns for different display devices. There may be generic color and black and white attributes, along with attributes specifically for certain types of displays (such as `XDEV_QDSS`, the color display on the Microvax II; see `/usr/include/X/X.h` for a listing of X-based display types). Programs that use **tap** will request attributes for a specific display (and `chromatism`, *i.e.*, is it a color or black and white display), if **tap** does not have information for that display, it will return either the `GENERIC-COLOR` (for color displays) or `GENERIC-BW` (for black and white displays). If the `GENERIC` attributes do not exist, **tap** will return any set of attributes that has the correct `chromatism`.

A color device such as a color PostScript printer may need its special patterns to get good quality output. The tool **oct2ps** also takes layer display information from **tap**.

To run **techBuild**, first pick a technology name (*TechName*). If necessary, create the technology directory

`% mkdir $OCTTOOLS/tech/TechName`

You can also create this directory anywhere else, and then you can use the `-=T` option to tell the tools where to find it. Let's assume that your pattern file for color output is called `tech.pat`, while your pattern file for black and white output is called `tech-bw.pat`.

Then you should run (possibly, the -T option should also be used, depending on the value of the `technology` in the pattern file):

```
techBuild -t TechName -v physical -C -O GENERIC-COLOR -L tech.pat
techBuild -t TechName -v symbolic -C -O GENERIC-COLOR -L tech.pat

techBuild -t TechName -v physical -C -O GENERIC-BW -L tech-bw.pat
techBuild -t TechName -v symbolic -C -O GENERIC-BW -L tech-bw.pat
```

You do not need to run **techBuild** in precisely this sequence. You should instead focus on one display type and one view type at a time, and go through the edit/run/check loop a few times until satisfied. For example, if you are defining the way the layers should look like in **vem** you will probably perform a few iterations of the following steps: edit the `tech.pat` file, run **techBuild**, start **a new vem** to look at the layers.

You can check what you have done with **tap-print** as well as with **techBuild**. For example

`% tap-print -l TechName physical`

shows the layer display information for physical views in the *TechName* technology. `techBuild -t TechName -v physical -l xxxx` gives you similar information (the last dummy arg is needed and ignored).

If you want to delete layer display information for a particular display type use

`% techBuild -D "GENERIC-BW" -t TechName -v viewname xxxx`

7.2 Design Rules

Currently **tech-edit** should be used to enter the design-rules about minimum and maximum width of a layer, layer to layer spacing, and layer to layer overlap.

Here are the sequence of operations necessary to run **tech-edit**:

1. start **vem**
2. open a window on the tap view

   ```
   "$OCTTOOLS/tech/TechName/tap.views:VIEWTYPE" : open-window
   ```

3. start tech-edit

   ```
   "$OCTTOOLS/bin/rpc/tech-edit" : rpc-any
   ```

4. to create a width rule (see Figure 7.2)
   (a) create a single box, with one dimension the size of the minimum width rule
   (b) put down two points, one on each edge of the box
   (c) `: min-rule`
5. to create a spacing rule (see Figure 7.2)
   (a) create two boxes, with the separation between the two boxes being the same as the spacing rule distance.
   (b) put down two points, on the inside edge of each box
   (c) `: min-rule`
   (d) if the two boxes are on the same layer, you get a spacing rule for objects on the same layer; if the two boxes are on different layers, you get a spacing rule for objects on the two layers.
6. in a similar way create maximum width rules and overlap rules.
7. exit tech-edit

   ```
   : exit
   ```

8. save the modifications to the tap view

   ```
   : save-window
   ```

9. exit **vem**

   ```
   : close-window
   ```

   (in the VEM console window)

Almost all tool in **Octtools 5.2** (e.g. **vem**, **MOSAICO**, **wolfe**, **sparcs**) use the design rules in **tap**.

7.3 Technology Properties

tap allows the storage and access of technology properties. These can be associated with a technology (as in Oct units per lambda), with a layer (as in sheet resistance), or with multiple layers (as in capacitance).

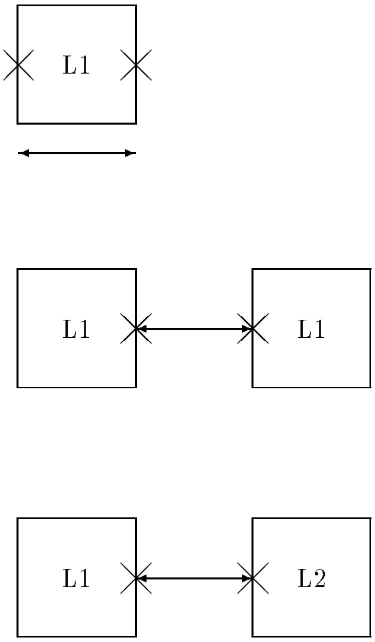
Technology properties can be entered into **tap** via **techBuild** and an ASCII file in yet another format: the `#` sign starts a comment that ends at the end of the line, and each property is described by a line formatted as:

`N {layer list} propname proptype propvalue`

where `N` is the number of layers associated with the property, and it will probably be 0,1,2, or 3, although any non-negative integer is allowed. `{layer list}` is a list of N layer names associated with the property. Following we have the property name, type, and value. Type can be either `R` for a real valued property, `S` for a string, or `I` for an integer.

Table 7.1 shows the technology properties defined for **Octtools 5.2**. Notice that the value of LAMBDA for the technology is determined by one of these properties. These properties are accessible procedurally through **tap** or **th**. Example of use of these properties can be found in the Parasitics Extraction Package **pep**.

Figure 7.2 shows a fragment of a technology property file. After preparing the file, or a set of files, you can transfer the information into **tap** with the command:



minimum width for a layer

spacing between objects a single layer

spacing between two layers (L1, L2)

Figure 7.1: How to Enter Design-Rules via Tech-Edit

N	Name	Type	units	description
0	LAMBDA	I		Oct units in one lambda
0	COORD-SIZE	R	m	
0	OXIDE-THICKNESS	R	m	
1	LAYER-CLASS	S		ROUTING, ANNOTATION, CONTACT, OTHER
1	SHEET-RESISTANCE	R	ohm/square	
1	CONTACT-RESISTANCE	R	ohm/m^2	Res. of a via of area $1m^2$
1	AREA-CAP-TO-GROUND	R	Farad/m^2	
1	PERIM-CAP-TO-GROUND	R	Farad/m	
1	THICKNESS	R	m	Thickness of layer over field
1	HEIGHT	R	m	Height of layer over field
2	AREA-CAP-TO-LAYER	R	Farad/m^2	
2	PERIM-CAP-TO-LAYER	R	Farad/m	

Table 7.1: Technology properties defined in **Octtools 5.2**.

```
# Tech properties for scmos

# this is a 1 micrometer per lambda technology (20x5e-8=1e-6)
0 LAMBDA I 20
0 COORD-SIZE R 5e-8     ## in meters

1 MET1 LAYER-CLASS S ROUTING
1 COPO LAYER-CLASS S CONTACT

2 MET1 MET2 AREA-CAP-TO-LAYER R 1e-6  # F/m^2
```

Figure 7.2: Fragment of a technolology property file.

```
% techBuild -t TechName -v symbolic filename
% techBuild -t TechName -v physical filename
```

The package **th** contains a procedure `thReadFile()` that allows programs to parse the content of a technology property file on the fly. The `tap.views` facet is temporarily modified for the duration of the process; all changes are visible only to the program and are discarded when the program exits.

7.4 Palettes

Palettes contain tool or package specific information. All palettes are instances that are attached to the facet of the particular tap view and contained by a bag called PALETTES, which itself is attached to the facet. Many of the **Octtools 5.2** expect all instances to be attached to the INSTANCES bag and to the facet. Some of those tools (e.g. **vem**, **octmvlib**, **alba**) may be useful to manage the `tap.views` facets.

Palettes named `"NAME"` will appear when you issue:

```
"NAME" : palette
```

in **vem**.

Important performance notice: if you want to improve the overall performance of **tap**, make sure that you instantiate the **interface** facet of your palettes in `tap.views`. This is particularly true of the connector palette and of any other palette that contains many instances (it is not an issue with the layer-palette). For example, assume that your connector palette contains 400 connectors, where each connector is an instance of a different master. If you instantiate the contents facet of this palette and then open the `tap.views` facet with **vem**, you will have to wait until **oct** has checked all 400 connector masters before **vem** will even start showing the facet. This entails about 800 file accesses, and it may take several minutes (which would seem like forever), especially when there is a network in between.

7.4.1 Layer Palettes

In order to make entering layout easier with **vem**, a layer-palette can be created. Using **vem**, create the cell:

```
$OCTTOOLS/tech/TechName/tap.views/layers:VIEWTYPE
```

where VIEWTYPE is the view of the technology (e.g. "physical" or "symbolic").

Important: Even when you edit the "symbolic" layer palette you need to be in the PHYSICAL edit style, or else **vem** will not allow you to enter boxes, but you will have to change the EDITSTYLE to SYMBOLIC when you are done (use the edit-property command in **vem**).

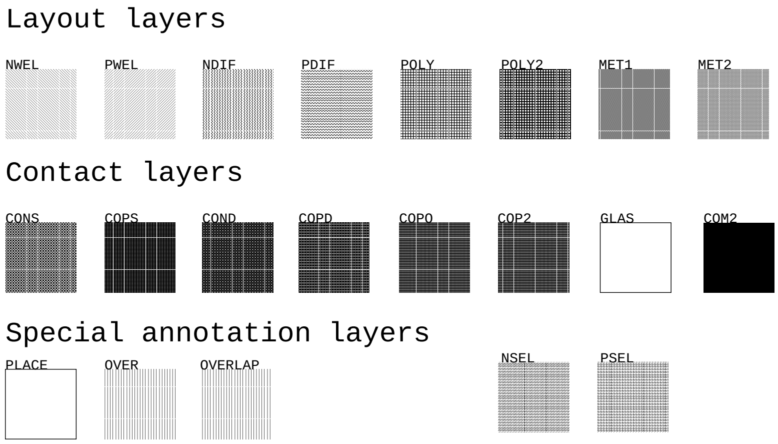


Figure 7.3: Example Layer Palette

In the palette, create by hand an array of boxes, one on each layer (see Figure 7.3). Arrange the boxes according to your own taste, and you may add labels to improve readability of the palette. Look at Figure 7.3 for an example.

Instantiate the layer cell in the tap.view cell:

```
$OCTTOOLS/tech/TechName/tap.views:VIEWTYPE
```

with the name 'layer' and attach it to the `PALETTES` bag. Now, this palette will appear when you issue the:

```
: palette
```

command in **vem**.

7.4.2 Connector Palettes

A connector is used to specify the connection of two different layers or a bend or t-junction of same layers (see Figure 7.4.2). A connector palette is a collection of connectors. A same layer connector is a single box on the layer, centered at (0,0) that implements a formal terminal 't'. The minimum size for a same layer connector is a box with sides the size of the minimum width of the layer. A multi-layer connector is composed of the layers to be connected and the via layers needed for connecting the layers. For example, a polysilicon to metal connector would contain geometry on the polysilicon, metal, and poly-metal via layer. Each of the connected layers contains a single box, centered at (0,0), and the via layer contains one or more boxes (depending on the process and design-rules). All of the geometries must implement a single formal terminal, 't'. All boxes on the connected layers should be the same size.

Due to the way symbolic layout is represented, in theory we would need to have an infinite library of connectors to support any legal connection between two wires of arbitrary width. Luckily, 95% of wires are minimum size, and for the other wire widths we can reasonably define a quantization sequence and an upper bound on width. Nevertheless, we may still need at least a few hundred connectors.

See **prim** for a program that creates connector instances for the *scmos* technology. Some tools use **prim** to generate special purpose connectors on the fly.

In *scmos* we put all connectors in the same directory:

```
$OCTTOOLS/tech/TechName/contacts
```

If you do not use **prim**, you can use **vulcan** to create the interface and then use **attache**, **vem**, **bdnet**, or **octprop** to put a property named `CELLTYPE` with value `CONTACT`, and a property named `CELLCLASS` with value `LEAF`, on the interface.

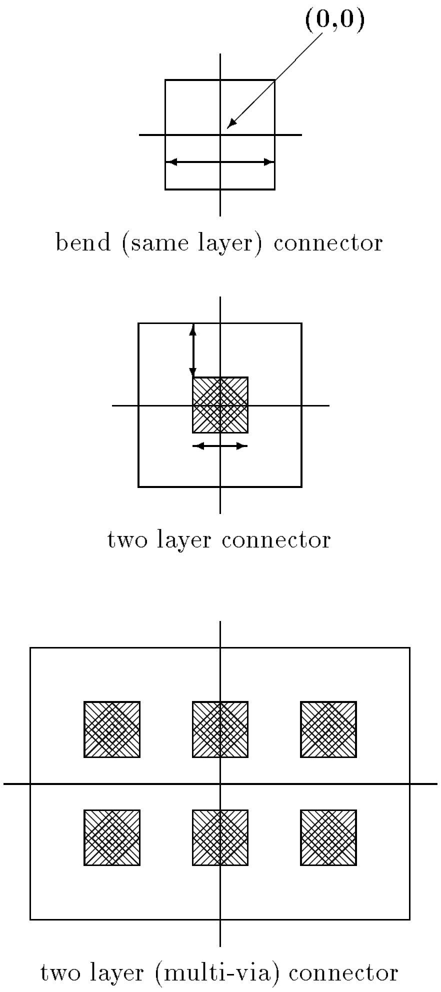


Figure 7.4: Single and Two Layer Connectors

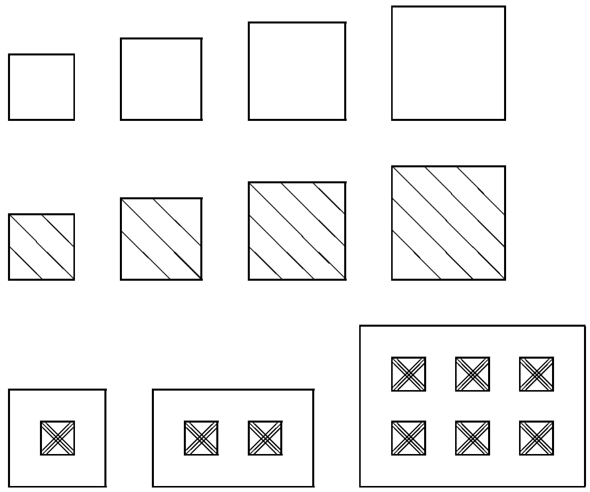


Figure 7.5: Example Connector Palette

```
% vulcan -k via:physical
% octprop -p "CELLTYPE=CONTACT" via:physical:interface
```

Once you have all connectors, the easiest way to create the connector palette is to use **connPalette**, which takes as input a file with a list of all connectors. **connPalette** also takes care of running **tapPP** (to do some performance enhancing pre-processing on the palette) and of instantiating the interface facet of the new palette in the appropriate `tap.views`.

This palette is used by **vem** (via **tap**) to determine the proper connectors when doing `SYMBOLIC` and `SCHEMATIC` entry. The palette is also used by tools and packages that create `SYMBOLIC` and `SCHEMATIC` designs (**symlib**, **wolfe**, **mosaico**, *etc.*).

In conclusion, the steps to create a connector palette are:

- Create all the connectors that you think will cover 90% of needs. You can use **prim**, or modify it depending on what is required by your technology. Some connectors can also be created by hand.
- List all connectors in one file and then run **connPalette**.

7.5 Testing a Technology Description

You can test the technology by using any of the tools or packages that use the technology descriptions, such as **vem**, **wolfe**, **mosaico**, **octpla**, **oct2ps**, **vulcan**, **symlib**, **pep**. You can also use a tool that was specifically created to check a technology description, **tap-print**

To verify the layer display information:

```
% tap-print -l TechName physical
% tap-print -l TechName symbolic
```

To verify the design-rules

```
% tap-print -d TechName physical
% tap-print -d TechName symbolic
```

To verify the palettes

```
% tap-print -p TechName physical
% tap-print -p TechName symbolic
```

And finally, to verify the actual connectors:

```
% tap-print -c TechName symbolic
```

7.6 Mask manipulation scripts

To complete a technology description you need to provide a sufficient number of **octmm** scripts, i.e. those scripts that perform mask manipulation. These scripts are normally installed in

`$OCTTOOLS/tech/TechName/octmm-scripts`

. Please refer to the manual page for **octmm** (`octman octmm`) for a detailed description of the syntax to be used in these scripts.

The tools that require these scripts are:

fabprep: is the tool that creates all the masks to be sent to MOSIS.

octdrc: is the "not so beautiful" design rule checker. Its scripts require a good amount of cleverness. The design rules need to be hardcoded into the script, because there is no way to get them from **tap**.

oct2mag: OCT to MAGIC converter.

mag2oct: MAGIC to OCT converter.

7.7 Special tools

At the date of this writing, the tool **kahlua**, that extracts a symbolic layout from a physical view, requires a special `design_rules` file (`octman kahlua` for more details). It is expected that **kahlua** will become conformant with the other tools, that is that it will also start using **tap**.

The compactor **sparcs** still accepts design rule files in the old format (`octman sparcs` for details). But since **sparcs** is now well integrated with **tap**, you should not worry about what that old format was.

7.8 Creating a Cell Library in Oct

There are a number of tools in the **Octtools 5.2** distribution that use cell libraries. This document describes how to convert a cell library from a foreign format into OCT.

7.8.1 Before Conversion

Create a physical technology description for the technology used by the cell library (see Section 7).

7.8.2 Converting Cells

Once a technology is created, you can start converting cells from your format to OCT physical views. The cell library should be located in a subdirectory of the technology directory. The Octtools distribution currently has converters for `Magic`[2] (**mag2oct**), and `CIF`[4] (**ciftooct**) format cells. Note that most formats do not have as much information as OCT maintains so some information will probably have to be added after the conversion. In particular, terminals usually get lost in the conversion (and in most cases never existed; `CIF` has no terminal construct), use **vem** to replace (or create) them. Make sure the terminals are implemented on one of the interconnect layers (**wolfe** requires that the signal terminals are implemented on the *vertical* layer defined by the `wolfe.rules` file, and is usually MET2).

In the `scmos` technology there are 20 OCT units per lambda, where lambda is some "technology-independent" parameter

7.8.3 Converting CIF Cells

The program **ciftooct** converts `CIF` cells to Oct. **ciftooct** uses a file that describes how to do the mapping. The mapping file consists of two components: a scale factor and a set of layer mappings. Layer mappings include logical operations and grow/shrink [2].

The first step is to determine a scale factor. There are 100 `CIF` units per micron. If you want your value of lambda to correspond to 1 microns you would scale that data by:

$$\frac{100\,CIF\ units}{micron} \times \frac{1\ micron}{lambda} \times \frac{lambda}{20\,Oct\ units} = \frac{100}{20} \tag{7.1}$$

Next, decide upon a layer mapping from `CIF` to Oct; if your Oct layers correspond to the `CIF` layers, then it is a trivial one-to-one mapping.

The following example scales a cell and does a one-to-one mapping of three layers:

```
#
#  example trivial mapping file
#

# set required Oct properties
property VIEWTYPE = PHYSICAL;
property EDITSTYLE = PHYSICAL;
property TECHNOLOGY = scmos;

# Lambda is 1 micron
options scale=20/100;

Layer1 = Layer1;
Layer2 = Layer2;
Layer3 = Layer3;
```

The next example converts a MOSIS PWELL process into the Oct layers for the scmos technology:

```
#
#  MOSIS P-well layers to Oct SCMOS Layers
#

property VIEWTYPE = PHYSICAL;
property EDITSTYLE = PHYSICAL;
property TECHNOLOGY = scmos;

options scale=20/150 infer_terminals;

labels MET1 = CMF;
labels MET2 = CMS;
labels POLY = CPG;
labels PDIF = CSP;
labels NDIF = CAA;

PWEL = CWP;
NDIF = CAA - CSP;
PDIF = CAA * CSP;
NWEL = grow(NDIF - CWP, 20);
COPD = (CCA * PDIF) - CWP;
COPS = (CCA * PDIF) * CWP;
COND = (CCA * NDIF) * CWP;
CONS = (CCA * NDIF) - CWP;
MET1 = CMF;
MET2 = CMS;
POLY = CPG;
COPO = CCP;
COM2 = CVA;
GLAS = COG;
```

[2] **ciftooct** uses the **octmm** package for this, see the **octmm** man page for a detailed description of the file format.

```
delete CWP CAA CSP CCA CMF CMS CPG CCP CVA COG;
```

Terminals

For each terminal, create a box on each layer that you want to use to implement the terminal (if you plan to use **wolfe** you must make sure one of the layers is the **wolfe** vertical interconnect layer). All boxes that implement the terminal must have common centers. Next a terminal must be created that contains the boxes. The easiest way to do all of this is in **vem** (use the `create-geometry` command to create the boxes and the `create-terminal` command to create the terminal).

Creating Interfaces

Once the cells have been converted and the terminals fixed, interface facets must be created for each cell. Use **vulcan** to create the interfaces:

```
% vulcan -k cell:physical
```

Use **vem** to make sure that the cells look correct before proceeding.

7.9 Extra Annotation for Tools

Depending on what you want to use the cells for you will need to add extra information. The following sections describe what extra information is needed for the tool described in the section. Note that the properties `TECHNOLOGY`, `EDITSTYLE`, and `VIEWTYPE` are required on all facets of all views of all cells. The properties `CELLCLASS` and `CELLTYPE` are required by a large number of tools. **vulcan** will propagate `TECHNOLOGY` and `VIEWTYPE` from the contents facet to the interface, and will create `EDITSTYLE` and `CELLCLASS` automatically.

7.9.1 musa

In order to use **musa** to do a logic-level simulation of the output of **misII** you must add extra information to the interface facets of the cells. The extra information consists of cell information and terminal information. The cell information is the `CELLCLASS` and `CELLTYPE`. `CELLCLASS` is a property that must have the value `LEAF`. `CELLTYPE` is a property that must have the value `COMBINATIONAL` for logic gates and `MEMORY`[3] for memory elements. Terminal information can be specified by attaching the `TERMTYPE` property to the terminal with one of the following values: `SIGNAL`, `SUPPLY`, `GROUND`, `CLOCK`, attaching the `DIRECTION` property[4] to the terminal with one of rhte following values: `INPUT`, `OUTPUT`, `INOUT`, and by attaching the `LOGICFUNCTION` property to each of the output terminals. The easiest way to place this extra information on the interfaces is by using **bdnet**. **bdnet** has two purposes: one is to connect blocks together, and the other is to annotate existing OCT facets. The following example shows the use of the **bdnet** annotation feature for adding the extra information:

```
!
! normal logic gate
!   shows use of permutable bag
!
(FACET 1120:physical:interface
    (SPROP "EDITSTYLE" "PHYSICAL")
    (SPROP "VIEWTYPE" "PHYSICAL")
    (SPROP CELLCLASS "LEAF")            ! a primitive
    (SPROP CELLTYPE "COMBINATIONAL")    ! a logic gate
    (SPROP "TECHNOLOGY" "scmos")
```

[3] **musa** only supports two very simple models of latches: non-resettable transparent and master-slave latches. This is being worked on. [JUST FIXED]

[4] Terminal directions are only important for terminals of termtype `SIGNAL`.

```
    !
    ! logic function format:
    !
    ! function:   "(" operator argument+ ")"
    ! operator:   "+" (or), "*" (and), "^" (xor), "!" (not)
    ! argument:   function | input-terminal-name
    !
    (FTERM O
        (SPROP LOGICFUNCTION "(! (+ 1A 1B))")
        (SPROP DIRECTION "OUTPUT")
        (SPROP TERMTYPE "SIGNAL"))
    (FTERM 1A
        (SPROP DIRECTION "INPUT")
        (SPROP TERMTYPE "SIGNAL"))
    (FTERM 1B
        (SPROP DIRECTION "INPUT")
        (SPROP TERMTYPE "SIGNAL"))
    (FTERM "Vdd!"
        (SPROP TERMTYPE "SUPPLY"))
    (FTERM "GND!"
        (SPROP TERMTYPE "GROUND")))

!
! two gates in a single cell
!
(FACET 1100:physical:interface
    (SPROP "VIEWTYPE" "PHYSICAL")
    (SPROP "EDITSTYLE" "PHYSICAL")
    (SPROP CELLCLASS "LEAF")
    (SPROP CELLTYPE "COMBINATIONAL")
    (SPROP "TECHNOLOGY" "scmos")
    (FTERM O1
        (SPROP LOGICFUNCTION "(! 1A )")
        (SPROP DIRECTION "OUTPUT")
        (SPROP TERMTYPE "SIGNAL"))
    (FTERM O2
        (SPROP LOGICFUNCTION "(! 2B )")
        (SPROP DIRECTION "OUTPUT")
        (SPROP TERMTYPE "SIGNAL"))
    (FTERM 1A
        (SPROP DIRECTION "INPUT")
        (SPROP TERMTYPE "SIGNAL"))
    (FTERM 2B
        (SPROP DIRECTION "INPUT")
        (SPROP TERMTYPE "SIGNAL"))
    (FTERM "Vdd!"
        (SPROP TERMTYPE "SUPPLY"))
    (FTERM "GND!"
        (SPROP TERMTYPE "GROUND")))
```

MOSFETS

musa is also a switch-level simulator and can handle circuits containing MOSFETS and logic cells. In order for **musa** to properly understand MOSFETS, they must have the following annotation:

terminals must have three terminals (can have more), named `DRAIN`, `GATE`, and `SOURCE`.

interface must have the following properties:

> `CELLTYPE: MOSFET`
>
> `CELLCLASS: LEAF`
>
> `MOSFETTYPE:` one of `NENH`, `PENH`, `NDEP`, or `PDEP`
>
> `MOSFETWIDTH:` width of the mosfet (in meters)

MOSFETLENGTH: length of the mosfet (in meters)

```
(FACET mosfet:physical:interface
    (SPROP "EDITSTYLE" "PHYSICAL")
    (SPROP "VIEWTYPE" "PHYSICAL")
    (SPROP CELLCLASS "LEAF")
    (SPROP CELLTYPE "MOSFET")
    (SPROP "TECHNOLOGY" "scmos")
    (SPROP MOSFETTYPE "NENH")
    (RPROP MOSFETWIDTH 0.000004)
    (RPROP MOSFETLENGTH 0.000004))
```

7.9.2 misII

In order to use **misII** to map a minimized logic network to your cell library you must create a `genlib` file. The `genlib` file describes the cells in the library: the name, terminals, logic functions, and delays[5]. The document `$OCTTOOLS/doc/genlib.doc` describes the format.

The following is an example `genlib` file:

```
GATE "1310:physical" 16 O=!1A;
PIN * INV 1 999 1 .2 1 .2

GATE "1120:physical" 24 O=!(1A+1B);
PIN * INV 1 999 1 .2 1 .2

GATE "1130:physical" 32 O=!(1A+1B+1C);
PIN * INV 1 999 1 .2 1 .2

GATE "1140:physical" 40 O=!(1A+1B+1C+1D);
PIN * INV 1 999 1 .2 1 .2

GATE "1220:physical" 24 O=!(1A*1B);
PIN * INV 1 999 1 .2 1 .2

GATE "1860:physical" 40 O=!((1A+1B)*(2C+2D));
PIN * INV 1 999 1 .2 1 .2

GATE  "1430:physical" 8 O=CONST1;
GATE  "1440:physical" 8 O=CONST0;
```

In order to read and write OCT versions of your logic networks using your cell library you must put on the annotations described in the **musa** section.

7.9.3 wolfe

In order to use **wolfe** to do a standard-cell place and route you must add information about electrically connected terminals and logically equivalent terminals. The **musa** example of **bdnet** use showed how this could be done. As mentioned at the beginning, make sure the terminals of the standard-cells are implemented on the *vertical* interconnect layer. **wolfe** needs a `SYMBOLIC` view type technology description. The technology description must contain a connectors palette and minimum width and spacing design-rules. See the document on creating technologies for a description of how to create one. **wolfe** also requires a `wolfe.rules` file that among other things describes the interconnection layers and the location of the feedthrough cell. See the **wolfe** man page for information on how to create the `wolfe.rules` file and the feedthrough cell. It is recommended that this file be located in the directory that contains the cell library. The following is an example `wolfe.rules` file:

[5] In a future version of **misII** the information in the `genlib` file will be stored in a **tap** palette.

```
units_per_lambda 20
h_layer MET1
v_layer MET2
h_net_weight 1.0
v_net_weight 1.0
rowSep 1.40
feedThru $OCTTOOLS/tech/scmos/msu/stdcell2_2/feed physical
power_width 16
power_position LEFT
```

Here is a sample **bdnet** file for annotating a feedthrough cell:

```
(FACET feed:physical:interface
    (FTERM "GND!" (SPROP TERMTYPE "GROUND"))
    (FTERM "Vdd!" (SPROP TERMTYPE "SUPPLY"))
    (BAG JOINEDTERMS
        (BAG JOINED
            (FTERM TOP)
            (FTERM BOTTOM))))
```

There are some other rules that apply to **wolfe**. The routing is performed on a grid with a pitch of 8 λ, which is the minimum pitch at which MET1-MET2 connectors can be placed on a line. It is therefore important that the cells are designed so that the terminals end up on the grid lines.

7.9.4 padplace

Placement of pads is handled by the **padplace**. Please refer to its man page for more information on how to describe pad families.

Chapter 8

Berkeley Logic Interchange Format (blif)

The goal of BLIF is to describe a Boolean network in textual form. A Boolean network is a representation of an arbitrary combinational logic network, and is an acyclic directed graph with a logic function attached to each node. Each node in this representation has a single output. Therefore, each net (or signal) has only a single driver, and we can therefore name either the signal or the gate which drives the signal without ambiguity.

In this section, angle-brackets surround nonterminals, and vertical bars separate choices in a pseudo-BNF style. Bdsyn uses only the *logic-gate* construct of BLIF. Bdsyn uses the the keywords *.module, .inputs, .outputs, .end,* and *.names.* The *module-reference* construct of BLIF is not implemented.

8.1 Modules

A module is declared by:

```
.module <decl-module-name>
.inputs <decl-input-list>
.outputs <decl-output-list>
<logic-gate> | <pla-reference> | <module-reference>
          .
          .
          .
<logic-gate> | <pla-reference> | <module-reference>
.end
```

decl-module-name is a string giving the name of the module.

decl-input-list is a white-space-separated list of strings (terminated by the end of the line) giving the formal parameters for the module being declared. If this is the last or only module, then these signals can be identified as the primary inputs of the circuit. Multiple *.inputs* lines are allowed, and the lists of inputs are merely concatenated.

decl-output-list is a white-space-separated list of strings (terminated by the end of the line) giving the formal parameters for the module being declared. If this is the last or only module, then these signals can be identified as the primary outputs of the circuit. Multiple *.outputs* lines are allowed, and the lists of inputs are merely concatenated.

It is expected that a BLIF parser may allow the *.inputs* and *.outputs* statements to be optional. If they are not specified, the primary inputs can be inferred from the signals which are not the outputs of any other logic block, and the primary outputs can be inferred from the signals which are not the inputs to any other blocks. This makes the serious assumption that no primary output is used as an intermediate signal. Also, these assumptions are only useful for the "root" module. Subcircuits are invoked with positional arguments and hence the specified order for the inputs is significant in a subcircuit.

The keyword *.end* is optional, and is implied at the end of the file.

Anywhere in the file, a '\' (backslash) as the last character on a line indicates concatenation of the subsequent line to the current line.

8.2 Logic Gates

A *logic-gate* is:

```
.names <in-1> <in-2> ... <in-n> <output>
<single-output-cover>
```

single-output-cover is an n-input, 1-output PLA description of the logic function corresponding to the logic gate. {0, 1, –} is used in the "input plane" and {0, 1, –, ~} is used in the "output plane". The logic gate can have only its ON-set specified with 1's in the "output plane", or can also have a DC-set and an OFF-set specified with -'s or 0's in the "output plane". ~ means no connection. For a more complete description of the PLA input format, see *espresso(5)*.

8.3 Module (subcircuit) references

A *module-reference* is

```
.subckt <module-name> <input-list> <output-list>
```

module-name gives the name of a module being included. It must be previously defined in the file. *input-list* are the actual parameters and are matched one-to-one with the corresponding *decl-input-list* inside the module declaration. Likewise, *output-list* are the actual parameters and are matched one-to-one with the corresponding *decl-output-list* inside the module declaration.

input-list and *output-list* are white-space separated sequences of tokens ending with the end of the line. A "/" may be used to separate *input-list* from *output-list*. The end of *input-list* and the start of *output-list* is, however, always inferred from the number of inputs and outputs in the module declaration (i.e., from *decl-input-list* and *decl-output-list* of the module which is being called).

If both *input-list* and *output-list* are not given, then the formal parameters within the module declaration become the actual parameters. It is an error if the *input-list* and *output-list* are given and the number of inputs or outputs does not match the "call" and the definition.

8.4 PLA References

A *pla-reference* is

```
.pla <pla-name> <input-list> <output-list>
```

pla-name gives the name of a file containing a PLA description in the format understood by *espresso(5)*. *input-list* and *output-list* are matched one for one with the inputs and outputs of the PLA (which, for arcane reasons are given via .ilb and .ob keywords within the PLA). Similar to *.subckt*, a "/" may separate the input-list and the output-list and if both are missing, the formal parameters become the actual parameters.

Index

Bibliography

[1] Xerox Corporation. Courier: The remote procedure call protocol. Technical report, Xerox Corporation, Stamford, Conn, February 1982.

[2] Gordon Hamachi, Robert Mayo, John Ousterhout, Walter Scott, and George Taylor. 1985 vlsi tools: More works by the original artists. Technical Report UCB/CSD 85/225, University of California, Berkeley, 1985.

[3] David S. Harrison, Peter Moore, Rick L. Spickelmier, and A.R. Newton. Data Management and Graphics Editing in the Berkeley Design Environment. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 24–27. IEEE, November 1986.

[4] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[5] SUN Microsystems. Remote procedure call specification. Technical report, SUN Microsystems, 1986.

[6] Robert Scheifler, James Gettys, and Ron Newman. *X Windows System: C Library and Protocol Interface*. Digital Press, Boston, Massachusetts, 1988.