OrCAD® X Capture Information Model

Product Version 23.1
September 2023

Document Last Updated: October 2019

Contents

1

Express Modeling Language

This document presents an information model of the objects of the database and their relationships to one another. These objects represent objects the electrical engineer works with in the creation of electrical designs.

Introduction

Electrical designs begin with the construction of primitive electrical parts and symbols. These parts and symbols are used in the construction of more complex electrical objects described through schematics. The designer may also create some graphics to aid the human readability of the schematics. The final complete electrical design may require numerous schematics.

During the design process, the user may build up libraries of reusable parts and schematics that will be useful in other designs. The objects modeled by our database correspond to the entities the electrical engineer creates in his work: Schematics, Pages, Libraries, LibParts, Packages, and Symbols. These basic building blocks are expanded to model the completed, computed hierarchical design and its backannotations. It also defines other reusable objects such as exported blocks of schematic connectivity and reusable graphic object. See Figure 1.

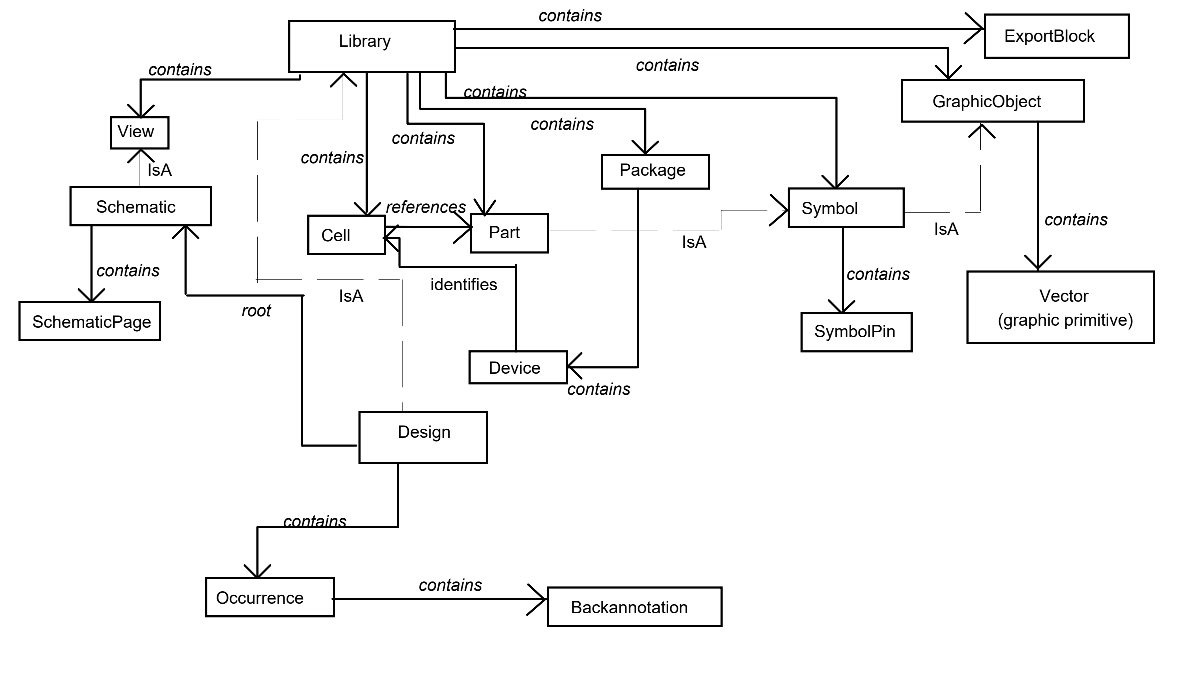


Figure 1: Overview of Objects

Overview of EXPRESS modeling language

The information model described below begins the process of defining an information model of the entities managed by our products. Each object is described in EXPRESS, a formal information modeling language. That language permits a complete description of the entities and their attributes. Relationships among entities are also described through attributes

Each object described in EXPRESS is enclosed by

```
ENTITY <entity name>
         <object description>
     END_ENTITY;
```

EXPRESS supports object hierarchy definitions. The SUBTYPE OF (<parent entity name>) phrase indicates that this entity currently being defined inherits all the attributes of parent entity. In C++ terms it is "derived from" the parent class. The SUPERTYPE OF (<child entity name, ...>) lists all the entities which are derived from this one. In cases where the term ABSTRACT precedes either of these phrases, the entity is a "pure base class" in the C++ sense. It is not possible to actually create one of these objects; you can only create one of the specialized entities. So, for example, you cannot create a Wire since Wire is defined as ABSTRACT; you can only create a WireScalar, or WireBus.

Each attribute of the entity is described by a statement of the form

```
          <attribute name>:<entity or type name>;
          <attribute name>:LIST[m:n] OF <entity or type name>;
          <attribute name>:SET [m:n] OF <entity or type name>;
```

Types are primitive objects such as strings, integers, colors, points, ...Those attributes which indicate an entity rather than a type name on the right hand side represent relationships between objects. For example, a Page has an attribute defined by

```
          Owner:    Schematic;
```

This indicates that the Page is related to a Schematic and that this relationship is described by the term "Owner". The Schematic is the owner of the Page.

Some relationships are required. In the example above, every Page must have an Owner. An owning Schematic must be specified in order to even create a new Page. Some relationships are not required. These are indicated by the addition of the word OPTIONAL in the attribute description. For example, a PortInst need not be connected to a Wire, so its Wire attribute is optional

```
          Wire:     OPTIONAL Wire;
```

Some relationships are not one-to-one. The LIST and SET phrases describe such relationships. A LIST is always ordered. The objects in the list may be inserted at a specific position in the list and iterators will always traverse the list in the same order. A SET is not ordered. There is no way to specify an object's position in the collection, nor is there any guarantee that iterators will traverse the set in the same order from one traversal to the next. The [m:n] indicates the permitted size of the list. The m indicates that the minimum size. Most collections specify 0 as the low bound which indicates that the collection may be empty. An empty collection still exists as a collection (just as an empty paper bag is still a paper bag). If the low bound is 2, then there must always be at least 2 members of the collection to even create the entity. The n indicates the upper bound on the size of the collection. If n is specified as a '?', that indicates that there is no upper bound on the size.

Some relationships are derived or computed from others. Such relationships are identified by the DERIVE specification:

```
DERIVE
          PinCount: Int;
```

The PinCount of a PartInst cannot be set by the user. It is computed by counting the number of actual pins on the PartInst. The user changes the value of PinCount by creating or deleting PortInsts.

Some attributes are “constrained” to take on certain values or follow certain restrictions. Such constraints are defined within EXPRESS with the WHERE clause. The definition of the rules is normally fully defined within the language. I have not completed such definitions primarily because it would require more knowledge of the EXPRESS language than the typical reader can be expected to have. I have, therefore, tried to describe the constraints through comments. Comments are limited to a single line and are preceded by two dashes (--).

The following EXPRESS has not been compiled and, as a result, is probably not syntactically correct. I have used the EXPRESS language because I feel it provides a useful, compact mechanism for describing the objects that will be managed by the database even in this incomplete state.

This is an “information model” rather than a description of the final, implemented interface. This model has been bound to a C++ interface; the description of that interface is found in the header files supplied for the db.lib library. In general, the C++ objects, member attributes and functions will follow the information model very closely. There will normally be a C++ object that corresponds to every EXPRESS entity. There will be a Get<attributeName> for every attribute specified in the EXPRESS entity description. There will be a Set<attributeName> for almost all non-derived attributes. For those attributes which model m-to-n relationships, there will be New, Add, Insert, Remove and Delete functions to modify the lists or sets.

Programming Considerations

Status Code

All DB functions calls return a status code either as the return value of the function or as an parameter. This status is encapsulated in the DboState object. The caller should always check this code for success or failure after a function call. In some cases, it may be necessary to retrieve the actual error code or message string.

```
DboState status;
.....
// ****check if function failed
if (status.Failed()) {
          // **** issue error message
          const char* pMsg = status.Message();
          .....
}
```

Each error code has an associated message string whose ID in the resource file corresponds to the error code.

Note: Currently all message strings are preceded by a message identifier in brackets followed by three new lines. The user will probably want to strip off this introductory material. *It is probable that this extra information will be removed in later releases*.

The list of error codes is available in db.h and the messages can be found in db.rc.

Note: The DB does not trap exceptions other than those resulting from IO activity. The application should set up exception handlers to trap system errors such as out of memory or memory faults. It was assumed that the application would typically require such handlers for its own activities and that there should be no need to add a redundant layer within the DB functions. In addition, DB functions typically cannot provide adequate error messages since they do not understand the application context in which the failure occurred.

See dbostate.h for the complete description of this object.

Containers

Containers are objects whose primary function is to aggregate other objects. A container manages all access to its contents. All contained objects must be created through the container; they cannot be constructed directly. All contained objects are removed from memory when the container is removed. All contained objects are deleted from persistent storage when the container is deleted. Every object “knows” the object that contains it.

The two primary containers defined in this document are the Library and the Design. All other design objects are contained within one of these either directly or indirectly as part of another object within the container. Many of the attributes of such contained objects represent information related to the container. Hence, objects cannot be directly constructed. They are always created through a NewXXX() function on the container. This guarantees that the object is valid in the context in which it exists. For example, the LibPart must have a name and that name must be unique in the scope of all LibParts in the Library.

Even the Library and Design objects must be created from within the Session. The Session acts as the “root” of the container tree.

Iterators

Access to collections is through an "iterator". This is an object created whenever a user requests access to a collection. The Nextxxx() function on the iterator returns the members of the collection one at a time. Iterators normally return pointers to an object. In such cases, a NULL pointer indicates the end of the iteration. The Nextxxx() function also returns a status. The normal end of the iteration is signaled by a status code of IDS_DBOSTATE_AT_END although it is normally sufficient to simply check that a NULL object is returned.

An iterator can be constructed directly through its constructor or created through the container whose contents are being scanned. If the iterator is created through the container using the NewXXXIter() method, the iterator is created on the heap and the application receives a pointer to the new iterator. The application should always explicitly delete the iterator after the iteration is complete to avoid memory leaks.

Some iterators may include a “mode” which specifies a subset of the collection to be returned in the iteration. The example following creates an iterator over the symbols in the Library but returns ONLY the Ports. It skips over all the Globals, OffPageConnectors, ERC Symbols, etc.

```
DboLibSymbolIter* pIter = pLib->NewSymbolsIter(Iter::PORTS);
```

Specific iterators are normally declared within the header file defining the collection over which the iteration takes place. Hence, the DboPagePartInstsIter is declared in dbopage.h

Examples of Iterator Usage

Example of iterator definition

The DboLib is a container of Symbols. The NewSymbolsIter() function on DboLib returns a pointer to an iterator that traverses Symbols within the library.

```
////////////////////////////////////////////////////////
//NewSymbolsIter()
//        Return a new itrator over the Symbols in this Library.
//        The iteration is controlled by the mode:
//        mode=[ALL | BOOKMARKS | ERCS | OFFPAGES | PORTS | GLOBALS]
//        This will read in each of the Symbols of the specified mode that is not
     yet in memory
//              and return a pointer to its in-memory image.
//        If the Symbol is already in memory, it returns a pointer to that existing
     image.
//        Status should always return IDS_DBOSTATE_OK
///////////////////////////////////////////////////////
DboLibSymbolsIter *NewSymbolsIter(DboState &status, Iter::IterModeT
     mode=Iter::ALL);
```

The following is an example ot the declaration of an iterator class:

```
//////////////////////////////////////////////////////////////////////////
class   DboLibSymbolsIter: public Iter
//////////////////////////////////////////////////////////////////////////
{
friend class DboLib;

public:   // constructors
    DboLibSymbolsIter(DboBaseObject *source, Iter::IterModeT mode = Iter::ALL)
        :Iter(LIB_SYMBOL_ITER, source)
    {m_nMode = mode; m_position = ((DboLib *)source)->m_pSymbolDir-
     >GetStartPosition();};


public:  // retrieval functions

    DboSymbol  *NextSymbol(DboState &status);



public:  // implementation of base class functions

    virtual DboBaseObject* Next(DboState &status)

    { return (DboBaseObject *)(NextSymbol(status));};


private:    //data

    POSITION m_position; //nxt Package to return

          Iter::IterModeT m_nMode;// mode of iteration = ALL,GlOBALS, PORTS,
     OFFPAGES

 };
```

Example of Creating Iterator Through Container on Heap

Containers have functions to create new iterators of the form NewXXX(). These functions return pointers to iterators created on heap and those iterators must be deleted when the iteration is complete to avoid memory leaks.

```
          //------------------------------------------------------------------
     ------------------------

          // Examine all the instances on the page

          //------------------------------------------------------------------
     -----------------------

DboPagePartInstsIter* pIter = pPage->NewPartInstsIter();

DboPartInst *pInst;

while ((pInst = pIter->NextPartInst()) != NULL) {

          //----------------------------------------------------------

          // Examine all the pins on the current instance

          //---------------------------------------------------------
          DboPartInstPinsIter* pPinIter = pInst->NewPinsIter();

          DboPortInst* pPin;

          while ((pPin = pPinIter->NextPin()) != NULL) {

               // -- do something with the PortInst

          }

          delete pPinIter;

}

delete pIter;
```

Example Use of Iterator using constructor and Default mode

```
//----------------------------------------------------------------------

// Examine ALL symbols (ports, offPage Connectors, globals, ERC markers...) in the
     library

//----------------------------------------------------------------------

DboLibSymbolsIter SymbolsIter(dboStatus);  // mode defaults to ALL

DboSymbol* pDboSymbol;


while ((pDboSymbol = pSymbolsIter.NextSymbol(dboStatus)) != NULL)

{

          // ------------------------------------

          // Do something with this port

          //------------------------------------

}

// don't need to worry about deleting iterator
```

Sample Use of Iterator - using mode

Mode is used to return a subset of an entire collection. The Library contains a set of Symbols. Symbols may be Ports, Globals, OffPageConnectors, Bookmarks, or ERC markers. Setting the mode can restrict the iterator to return only objects of 1 type. ALL will return all members of the list. In this example, the iterator is restricted to return only the Ports of the library through the mode.

```
//-----------------------------------------------------------------------

// Examine each PORT symbol in the library

//------------------------------------------------------------------------

DboLibSymbolsIter *pSymbolsIter = pDboLib->NewSymbolsIter(dboStatus, Iter::PORT);

DboSymbol* pDboSymbol;


while ((pDboSymbol = pSymbolsIter->NextSymbol(dboStatus)) != NULL)

{

          // ------------------------------------

          // Do something with this port

          //-------------------------------------


}

delete pSymbolsIter;
```

Memory Usage

Many functions return pointers to objects. Except for the Session, those objects are always "contained" in some container: a Library, Design, or Page. Generally such pointers will remain valid as long as that container remains in memory (DeleteXXX() or RemoveXXX () remove the container from memory) or that object itself is not deleted or removed.

There are some exceptions to this rule. Wires in the page may be deleted during a call to DboPage::Connect(). Wires that connect end-to-end or share segments are merged at that time. Nets are also deleted and recomputed at that time to represent the new connectivity. See BeginTransaction() and CommitTransaction() described for the Schematic Page section for more information.

SchematicNet, SchematicPort, Occurrence and FlatNet pointers become invalid if the underlying schematic pages change. These computed objects are always computed only upon request and must be rebuilt after the page is edited; any pointers to the previous computations must be discarded. See the sections describing the Schematic object and its contents for more information.

NetOccurrence and PortOccurrence pointers also become invalid if the underlying schematic pages change. These objects are also computed only upon request and must also be rebuilt after the page is edited.

Persistent Objects

Most of the objects managed by the DB are “persistent”: they have an image on disk which permits retrieving the object repeatedly over time. Only the two largest containers, the Library and Design, correspond to physical files. These containers are implemented as OLE 2.0 Compound Files. All other objects are contained within these as other IStorage objects or as IStreams.

Larger objects or collections can normally be read from disk independently of other objects. Other objects cannot be read in independently of their container; it is not possible to get a single Instance from a page without reading in the entire page. The following objects can be read and saved independently:

- **Library**: Reads in a directory of its contents and default values. It also reads in the cache of all symbols and packages which have been placed within schematics in the library. It does not read in the contained Packages, Symbols, Parts, Views or ExportBlocks. When Saved, it updates the disk image of any Package, Symbol, or Schematic for which a save has previously been requested.
- **Design**:Reads in its basic library data, and the name of its root View and its occurrence hierarchy. When Saved, it updates the disk image of the design hierarchy and any Schematics for which a save has been requested.
- **Package**: Reads in its Devices and related Cells and LibParts. When Saved, it updates the disk image of the related Cells and LibParts. One Package may be saved independently of any other Packages or Symbols.
- **Symbol**:Reads in its graphics and its SymbolPin definitions. Each symbol is saved independently of all others in the library.
- **GraphicObject**: Reads in its graphics. Each graphic object is read and saved independently of all others in the library.
- **Schematic**: Reads in a directory of its pages and the set of unique identifies already used within the schematic. It does not read in the contained Pages.
- **Page**: Reads in all its contained Instances, Wires, Nets, ...

Session::SaveLib or Session::SaveDesign saves the current state of a Library or Design to disk and leaves the object in memory. For other objects, “SaveXXX” DOES NOT write the object to disk immediately. It simply marks the object to be saved to disk whenever its containing Design or Library is written to disk. This permits the user to simultaneously edit several pages or parts in the Design or Library and update the disk image of only those he wishes. See the section describing the Library for more detail on the behavior of Save.

container->GetXXX(name) opens the specified persistent object and loads it into memory if it is not yet open, otherwise it returns a pointer to the existing object.

container->RemoveXXX(pObj) removes the object from memory when the application no longer needs it. It has no affect on the disk image.

container->DeleteXXX(pObj) removes the object from memory and marks it to be deleted from the disk image the next time its container is saved.

container->CopyXXX(pObj) makes an in memory copy of the object within the container; a new disk image is created when the containing Library or Design is saved.

Session Objects

Session

```
ENTITY Session
          Libraries:SET [0:?] OF Library;
          Designs:  SET [0:?] OF Design;
END_ENTITY;
```

The session provides the environment within which all design and library data is accessed. A Session must be constructed before any other design access function can be called. It is only through a Session that the application can Create, Get, Remove, Save or Delete a Library or Design. Normally an application will construct a single Session. A Windows application would normally provide access to that Session through the main App or through a global pointer.

See DboSessn.h

Basic Objects

BaseObject

```
ENTITY BaseObject
ABSTRACT SUPERTYPE OF (
                        Alias,
                        BusEntry,
                        Device,
                        DisplayProp,
                        FlatNet,
                        GraphicInstance,
                        Library,
                        LibObject,
                        Net,
                        Occurrence,
                        OptimizerParameter,
                        Package,
                        Page,
                        PortInst,
                        SchematicNet,
                        SchematicSymbolInst,
                        SymbolPin,
                        Vector,
                        View,
                        Wire)
          Type:           ObjectTypeT;
          DBProps:  SET [0:?] OF DBProp;
          PermitsUserProps:Boolean;
          UserProps:OPTIONAL SET [0:?] OF UserProp;
          DisplayProps:OPTIONAL SET [0:?] OF DisplayProp;
```

```
DERIVE
          TypeString:StringT;
          ContainingLib:Library;
          EffectiveProps:SET [0:?] OF (name, value) pairs;
END_ENTITY
```

The most basic object from which all others are derived. All BaseObjects have a type. The type may be returned as an enumeration or as a string. See dbobase.h for a list of all object types. All objects except the Session also have a ContainingLib. Since a Design is derived from Library, objects contained in a Design have that Design as their ContainingLib. Libraries and Designs contain themselves.

All objects also have a set of DBProps. These DBProps are the basic attributes of the object; the DBProps are identified within each of the entities defined in this information model. They are (Name, Value) pairs. Values are typed; they may have type String, Integer, Boolean, Time, Color, PinType, ... Although the value is typed, the application may always access/edit the value as a string through the DBProp::GetStringValue() and DBProp::SetStringValue() functions. An iterator permits accessing all the DBProps of an object or the application may look up a DB prop by name or by string ID. Valid IDs for the DBProps may be found in db.h as IDS_PROP_xxx definitions and the associated strings can be found in db.rc.

All objects may optionally have a set of UserProps. These are (Name, Value) pairs added to an object by the user. The Value is limited to a string. The properties provide the user a means of saving attributes for other tools which have no semantic meaning to this schematic capture application. Some objects may not permit UserProps to be added, in which case PermitsUserProps is FALSE. An application can iterate through all the UserProps of an object or look up a specific property by name.

An object may also have an optional set of DisplayProps. DisplayProps permit the visual display of UserProps or DBProps on the schematic page. An application can iterate through all the DisplayProps of an object or look up a specific property by name.

The EffectiveProps of an object are computed. There is no actual object modeling an “EffectiveProp”; there are “EffectiveProp” functions on the BaseObject which provide a simple interface to both the system and user added properties. The application need not know whether the attribute is a DBProp or a UserProp, only its name. The application also does not need to know the type of the value and may always work with properties as if they were string values. For most objects the set of EffectiveProps is simply the set of DBProps combined with any UserProps the user may have added. If the name of a UserProp is the same as that of a DBProp, the UserProp overrides the DBProp. Instances, Nets, and Occurrences compute their EffectiveProps in more complex ways which are described for each of these object in more detail later. An application can iterate through all the EffectiveProps of an object or look up specific properties by name. The application may also update UserProps and DBProps through the SetEffectivePropStringValue() function. This function will set the DBProp, UserProp or create a new UserProp as appropriate for the object and the specific property.

See DboBase.h

DBProp

```
TYPE ValueTypeT

ENUMERATION OF (STRING_T, INT_T, LONG_T, BOOL_T, COLOR_T, FONT_T, PIN_TYPE_T,
     PIN_STYLE_T, LINE_STYLE_T, LINE_WIDTH_T, FILL_STYLE_T);

END_TYPE;


ENTITY DBProp

          Name:     StringT;

          Value:    Value;

          ReadOnly:BooleanT;

DERIVE

          Type:     ValueTypeT;

END_ENTITY;
```

These (Name,Value) pairs define the most basic characteristics of the object. The DBProps provide a generic method of accessing the attributes of any object; they correspond almost identically to the entity attributes described in this document. Regardless of type, functions to Get and SetDBPropStringValues() permit interaction with this objects via strings only.

Some DBProps cannot be updated. Typically this indicates the information has been computed from some other source or is an identifier for the object. If the prop cannot be updated, ReadOnly is TRUE.

ID's are defined for all DBProps in db.h. The strings associated with these names are defined in the string table in db.rc. Property lookup functions accept either a name string or the string's ID.

See DbProp.h and DboValue.h

UserProp

```
ENTITY UserProp

          Name:     StringT;
```

```
        Value:    StringT;
END_ENTITY;
```

The UserProp permits the user to attach information to objects in a Design which are tool or design specific. Capture manages the information rather like a black box; it has no understanding of the semantics of these values. UserProp Values can be displayed on the SchematicPage through a DisplayProp.

See DboUProp.h

EffectiveProp

There is no actual EffectiveProp object. The effective properties of an object are accessed through functions on the object itself.

Most applications will use the EffectiveProps functions since it simplifies access to properties. The user need not worry whether the property is held in a DBProp or a UserProp or if there are rules for looking at multiple objects for the property. The default behavior for GetEffectivePropStringValue() is:

1. Look 1st for a UserProp on the object with the specified name.
2. If not found as a UserProp, look for a DBProp of the object with the specified name.

Some objects have much more complicated rules for computing the EffectiveProps. An InstOccurrence is an example of a specialized algorithm for getting Effective Prop values:

1. Look 1st on the InstOccurrence at backannotations held as UserProps on the Occurrence.
2. If not found, look at the DBProps of the InstOccurrence.
3. If not found, look at the UserProps of the PartInst which is the source of the Occurrence.
4. If not found, look at the DBProps of the PartInst.
5. If not found, look at the UserProps of the LibPart which defines the PartInst.
6. If not found, fail.

This permits the application to override any basic properties added to the LibPart with instance specific information and to override any instance properties with even more specific occurrence information.

Iterators over EffectiveProps will return all the UserProps and DBProps of the object and apply such rules for defining the set of properties as are used in the lookup. Hence, iterating through the EffectiveProps on an InstOccurrence will return the backannotations, the Occurrence's DBProps, the PartInst's UserProps, the underlying LibPart's UserProps, etc. If the same property exists on more than one of these objects, it will return the appropriate overriding value.

Example of Getting EffectiveProp Value

The value of an EffectiveProp is always returned as a string.

// Get the page number from the title block

CString cstrPageNumber = ((DboBaseObject*)pBlock) -> GetEffectivePropStringValue(IDS_PROP_DESIGN_PAGE_NUMBER,

status);

Example Use of EffectivePropsIter

The EffectivePropsIter is somewhat more complex than normal iterators since it does not return an object. Since it is not possible to query the object about itself, extra information is returned by the NextEffectiveProp() function:

```
          PropName:string which is name of property
          PropValue:string which is value of property in string form
          PropType:Value type return - see Value.h for enumeration
          bEditable:TRUE if the property can be edited, else FALSE
```

```
// Iterate over all the effective properties of the current device

DboEffectivePropsIter *pPropIter = pCurrentDevice->NewEffectivePropsIter(status);

while (pPropIter->NextEffectiveProp(PropName, PropValue, PropType,
     bEditable).OK()){

          // Skip over the props we do not want to output into the netlist.

          if (listIllegalProps.Find(PropName))

               continue;

          if (!mapProps.Lookup(PropName, strOriginalPropValue))

               mapProps.SetAt(PropName, PropValue);

} //  end while
```

```
delete pPropIter;
```

Example of Setting EffectiveProp Value

Setting the EffectiveProp will set the UserProp value if there is a UserProp with the specified name. If there is no UserProp, it will set the value of any DBProp with that name. If there is no existing DBProp with the specified name, it creates a new UserProp and sets its value. Note that the function may be called with either the property ID or with a string.

pPortInst->SetEffectivePropStringValue(IDS_PROP_IS_NO_CONNECT, newNCValue);

pPortInst->SetEffectivePropStringValue(CString("NewProp"), CString("NewValue"));

DisplayProp

```
ENTITY DisplayProp

SUBTYPE OF (BaseObject);

          Owner:         BaseObject;

          Name:          StringT;

          Location: Location;

          Rotation: RotationT;

          Font:          FontT;

          Color:         ColorT;

          DisplayType:DisplayT;

END_ENTITY;
```

The DisplayProp provides a means of graphically displaying a property of an object. The name of the DisplayProp specifies the name of the UserProp or DBProp whose Value is to be displayed. If there is no property on the object with that Name or the UserProp has no Value, the name of the UserProp/DisplayProp is displayed at the specified position.

```
The DisplayType controls the format of the display:
          VALUE_ONLY:only the value displays, the default behavior
          NAME_ONLY:  only the name displays
          NAME_AND_VALUE:  the property displays in the format <name>=<value>
          BOTH_IF_VALUED:  display as above only if the user has specified a value
     for the property
```

The Location is relative to the Owner object. This permits the DisplayProp to be attached to either a GraphicObject in the Library or an instance in the Page.

See dboprop.h

Note: The *Value If Value Exists* option will only work if the display type is set to `BOTH_IF_VALUED` and a mask is set to `MASK_VALUE_IFVALUE`. If mask is not set, *Value If Value Exists* option will not work.

Note: The property is locked using combination of the following commands:

```
$lObject    SetBitmask      $::DboBaseObject_MASK_LOCKED
$lObject    SetBitmask      $::DboBaseObject_MASK_TEMPLOCK
```

Vector

```
ENTITY Vector
ABSTRACT SUPERTYPE OF (Box, Line, Arc, Point, Polygon, Polyline, CommentText);
END_ENTITY;
```

A Vector is an abstract object which represents a graphic which can be manipulated by the application as a single entity.

Box

```
ENTITY Box
          UpperLeft:     Location;
          LowerRight     Location;
          LineStyle:     LineStyleT;
          LineWidth:     LineWidthT;
          IsSolid:       Boolean;
          FillStyle:     FillStyleT
          Color:         ColorT;
END_ENTITY;
```

See DboVect.h

Line

```
ENTITY Line
          Start:          Location;
          End:            Location:
          Color:          ColorT;
          LineStyle:      LintStyleT;
END_ENTITY;
```

See DboVect.h

Arc

```
ENTITY Arc
          Center:         Location;
          XRadius:  Long;
          YRadius:  Long;
          Start:          Location;
          End:            Location;
          LineStyle:LineStyleT;
          LineWidth:LineWidthT;
          Color:          ColorT;
END_ENTITY;
```

See DboVect.h

Polygon

```
ENTITY Polygon
          LineStyle:LineStyleT;
          LineWidth:LineWidthT;
          IsSolid:       Boolean;
          FillStyle:FillStyleT;
          Color:         ColorT;
          Points:        LIST [0:?] OF Location;
END_ENTITY;
```

See DboVect.h

PolyLine

```
ENTITY PolyLine
          LineStyle:LineStyleT;
          LineWidth:LineWidthT;
          Color:         ColorT;
          Points:        LIST [0:?] OF Location;
END_ENTITY;
```

See DboVect.h

CommentText

```
ENTITY CommentText
          Text:     StringT;
          Coordinate:Location;
          Rotation:RotationT;
          Font:     Font;
          Color:    ColorT;
END_ENTITY;
```

See DboVect.h

BitMap

```
ENTITY BitMap
          BoundingBox:   RectangleT;
          Size:          int;
          Dimension:     RectangleT;
          map            voidT;
          Coordindinate: Location;
END_ENTITY;
```

See DboVect.h

Library Objects

Library

```
ENTITY Library
SUBTYPE OF (BaseObject);
          Name:          StringT;
          TimeCreated:   TimeT;
          TimeModified:  TimeT;
          Packages: SET [0:?] OF Package;
          PackageAliases:SET [0:?] OF PackageAlias;
          Cells:         SET [0:?] OF Cell;
          Parts:         SET [0:?] OF LibPart;
          Views:         SET [0:?] OF Views;
          Symbols:  SET [0:?] OF Symbol;
          ExportBlocks:SET [0:?] OF ExportBlocks;
          Graphics: SET [0:?] OF GraphicObject;
          Cache:         SET [0:?] OF BaseObject;
          DefaultFontsLIST [0:?] OF FontT;
          DefaultPlacedInstIsPrimitive:BoolT;
          DefaultDrawnInstIsPrimitive:BoolT;
          FieldMap: LIST [0:?] OF StringT;
          UserStorageIStorage;
DERIVE
          IsPersistent:BoolT;
          IsModifiedBoolT;
          ChangedObjects:SET [0:?] of BaseObject;
END_ENTITY
```

Our customers use our tool primarily to create parts in a part library or schematic sheets in a design. The database models this by providing two major physical container objects, the Library and the Design. All other design data is contained within one of these.

A Library is a generic container of reusable design data. It contains LibParts which have been created to act as templates for instances on Design Pages. It also contains a variety of symbols such as PortSymbols, OffPageConnectorSymbols, and power and ground GlobalSymbols.

The LibParts of a Library may be associated by grouping them into Cells. Although there are no constraints on what a user puts into a Cell, the intent is that he regards all the LibParts in the Cell as electrically the "same". This grouping is not a containment; the LibPart is not owned by the Cell, but is “referenced” by the Cell. The same LibPart can be shared by multiple Cells.

The Library also contains Packages. The Package acts as a template for a physical package on a printed circuit board. It provides the relationship between the logical gate represented by the Instance on the page and the part on the board. A Package in a Library may be associated with several names. A Package may be retrieved using any of its aliases as well as its name.

Note: *The current release of Capture requires that a strict naming convention be followed*. *The first LibPart in a Cell must have a name of the form <baseName>.Normal*. *If a second part is added to a Cell, its name must have the form <baseName>.Convert*. The *basename must be that of the referencing Cell*. *It is not currently possible, therefore, to share Parts among Cells and the relationship is more one of containment.*

Note: In order to support the interactive editor, the DB currently treats the Cell and LibPart much as if they were contained in the Package object. They are read and saved as one unit. Use the SavePackageAll(), CopyPackageAll(), RemovePackageAll(), DeletePackageAll(), RenamePackageAll()... functions to save or edit the Package and all its referenced Cells and LibParts as one entity.

The user may create custom, reusable symbols as well as parts and packages within a library: Port symbols, OffpageConnector symbols, and Global symbols can be instanced on a page to change the characteristics of a net. Bookmarks can be placed to mark locations on a page and ERC markers mark errors detected in the design. The user may create custom Titleblock symbols which he reuses on all his designs. PSpice users may create a number of symbols that give instructions to the simulator when placed on a page: Parameters, Optimizer Parameters, Stimulus, Sources, Directives.

Reusable Schematics may be stored in a Library. Normally the Library would contain both the Schematic and a LibPart which referenced that Schematic as its “contents”. In future releases, other types of Views will also be supported by the Library.

The user may also save subsets of schematic information in ExportBlocks within a Library. These blocks are assumed to be reusable bits of connectivity that the user wishes to place as a single entity on numerous schematic pages in different designs.

Note: *The database currently provides for saving reusable graphics in the Library, but the UI does not yet support such objects*.

The relationship between the Library and the LibParts, Symbols, Packages, and Views it contains is similar to the relationship between a directory and the files it contains. The files are contained by the directory; the LibParts and Symbols are contained by the Library. If the directory is deleted, all the files are deleted; if the Library is deleted, its LibParts and Symbols are deleted. If the container is moved, all its contents are moved. No file can be created without being contained in a directory; no LibPart can be created without being contained in a Library or Design.

The functions to create, remove, save, and delete a contained object are always on the containing object. This is because the container must manage those objects it contains. The container/contained relationship does not imply the user must always work only at the container level. The user updates a single file independently of the directory or the other files within it. He saves the file, not the directory. Similarly, the user edits a LibPart or Symbol independently even though it is contained in the Library. Whenever an object in the Lib is modified, the object is marked as modified and the containing Library is also marked as modified. The application can query the Library about whether it has been modified and may iterate over the set of objects which have been changed since the last Save.

Saves are a two phase process: the objects are marked to be saved using the object’s Save() function and then the containing Library or Design is saved. It is the only the save of the highest level container that actually results in updating the disk image. In many cases the application may wish to synchronize the save of several objects. For example, Parts, Cells and Packages are all updated simultaneously during part editing in Capture. The edits to the Part should not be saved without the corresponding edits to the Package. In such cases, the application marks the objects for save and then saves the Library. In the case of a page, the application marks the Page to be saved, marks the containing Schematic to be saved, and then saves the Design or Library.

Because of the large number of LibParts contained in a typical Library, it is not reasonable to have a separate file or directory for each. For this reason, the Library object is a single file. Regardless of its physical representation as a single file, it is possible for the Library to load only those portions that are required to represent the desired LibParts; and it is be possible to save a LibPart in that file without updating all the LibParts which are currently loaded or being edited. The OLE CompoundFile provides support for this functionality. The diagram on the following page shows the physical structure of the Library's compound file. An IStorage is a container of other IStorages and IStreams. Although the Compound File is a single physical file on disk, logically the IStorage acts as a directory and an IStream acts as a file. It is possible to open an IStorage and iterate over its contents without reading the entire file or even any of its contained streams. The IStorages and IStreams within an IStorage can be opened/read/saved independently of one another. Due to the large number of file handles that CompoundFiles require, however, the DB does not attempt to leave these storages and streams open or use the transaction management they provide. See Figure 2 for a layout of the physical storage.

Note that objects are grouped into IStorages based on type: all packages are grouped into the "Packages" IStorage while all Views are grouped into the "Views" IStorage. This is necessary since it is common for some libraries to contain both a package and a schematic that represent the same electrical object and, hence, have the same name.

There are some limitations on naming as a result of using the CompoundFile structure to implement the Library object:

1. Names of IStorages and IStreams are limited to 32 characters. This means Views, Packages, ExportBlocks, and Symbols are also limited to 32 characters.
2. Names may not contain '/', '\', or ':'. This means that View, Package, ExportBlock, and Symbol names may not contain these characters.
3. Names are saved as entered, but comparisons are case insensitive. This means that you may create a Package named "Fred" and it will display as mixed case, but you will get an error that the name is in use if you try to also create a Package named "FRED".

There is currently no locking mechanism or verification that an update is on the latest version. User1 may get Part1 and begin editing. User2 may also get and edit the same Part1. User2 is not notified that the Part1 is already in use. User2 may then save his changes that would be in conflict with those of User1. User1 will not be notified that his edits are on an earlier version of the Part.

Whenever a LibPart is instantiated on a Page within a Library or Design, copies of the defining LibPart, Cell, and Package are saved in the Library's cache. All other instances of that part reference the same cached LibPart, Package, and Cell. Copies of Symbols are similarly cached whenever a Symbol is instantiated on a Page. If the original LibPart is altered, the cache is not automatically updated. The instances continue to reference the cached copy throughout the design. The user may request that a cached copy be updated to match a newer version or that it be replaced by a different part. The cache provides 3 functions:

1. Consistent use of a part throughout the design with controlled update points.
2. Performance improvement by avoiding opening multiple libraries to get parts repeatedly.
3. Archiving is not necessary. All parts are available within the design itself and external Libraries need not be accessed.

Note: Currently the cache does not contain copies of any external schematic, VHDL, or PSpice simulation models and hence cannot act as an archive in all cases

The database provides iterators over the objects cached in a Design or Library as well as functions to update a cached part to a newer version or replace the part with another.

A UserStorage area is provided for the use of applications to store additional data which is not directly managed by the DB. It is anticipated that this storage area will be used to hold related library data such as PSpice simulation information or CIS part data. The data within the storage is not available unless that application is running; standard Capture does not “understand” it. It is essentially a very large UserProp at the library level.

See dbolib.h

Example of Creating a New Library

The following shows the display of a dialog box to get the library name and the creation of a new library within the application's session.

```
// ***********Display the dialog Box**************

COrAttachSchDlg dlg(NULL, IDS_EXPORT_BLOCK, IDS_EXPORT_BLOCK_NAME);

if (dlg.DoModal()==IDOK)  {

          // ****** Get the session from the application ********

          DboSession *pSession = ((COrSdtApp *)AfxGetApp())->GetSession();

          // ****** Create a new library in memory with the user's specified path
      name**********

          DboLib *pLib = pSession->GetLib(dlg.m_szLibName, dboStatus);

          if(NULL == pLib){
               pLib = pSession->CreateLib(dlg.m_szLibName, dboStatus);
               if (dboStatus.Failed()) {
                    // process error
               }
          }
          // ******** lib already exists-> cannot create************
          else {
               // process error
          }
}
```

This only creates the DboLib in memory. Nothing has been written to disk until the library is Saved.

Example of Saving a Library Object

To save a library object, first mark the object to be saved and then save the library. This 2 phase save permits several objects that must be saved simultaneously to be marked and saved in one transaction. The SavePackageAll() function called below marks the Package, and all its associated Cells and Parts to be saved before the library is actually updated on disk.

```
//-----------------------------------------------------------------
// Mark the package and all associated Cells and LibParts to be saved in the
     library.
//-----------------------------------------------------------------
if (!(dboStatus = m_pDboLib->SavePackageAll(m_pDboPackage)).Succeeded())
                    ThrowDboException(dboStatus);


//-----------------------------------------------------------------
// save the library to disk and issue error if fails
//-----------------------------------------------------------------
if ((dboStatus = pSession->SaveLib(m_pDboLib)) == DBO_FAILED)
          {
          ThrowDboException(dboStatus);
          }
}
else  }
          .....
```

Example of Saving All Objects Which Have Been Edited

```
//-----------------------------------------------------------------------------
     -------------------------
//  Iterate through all objects that have been modified in the library
//-----------------------------------------------------------------------------
     -------------------------
DboLibChangedObjectsIter *pChangedIter = pDboLib-
     >NewChangedObjectsIter(dboStatus);
DboBaseObject *pChangedObj = NULL;
while ((pChangedObj = pChangedIter->NextObject(dboStatus)) != NULL)
{
          int nType = pChangedObj->GetObjectType(dboStatus);
          //------------------------------------------------------------------
          // Mark the object to be saved
          //-----------------------------------------------------------------
          pDboLib->SaveObject(pChangedObj);
}
delete pChangedIter;

//-----------------------------------------------------------------------------
     --------------
//  Save the Library to flush all edits to disk
//-----------------------------------------------------------------------------
     -------------
if ((dboStatus = pSession->SaveLib(pDboLib)) == DBO_FAILED)
{
          ThrowDboException(dboStatus);
}
```

LibObject

```
ENTITY LibObject
SUBTYPE OF (BaseObject);
ABSTRACT SUPERTYPE OF (  Cell,
                    GraphicObject,
                    Package,
                    View);
          TimeCreated:   TimeT;
          TimeModified:  TimeT;
          Owner:    Library;
          Name:     StringT;
DERIVE
          IsModified:    BooleanT;
          IsPersistent:  BooleanT;
          IsCachedCopy:  BooleanT;
END_ENTITY
```

The LibObject is an abstract class. It defines the basic attributes of all objects directly owned by a Library. These objects are independently readable, savable objects. They all have time stamps which indicate when their persistent image was last updated. LibObjects have unique names within a name space depending on object type:

- Views of all types share the same name space
- Packages share the same name space
- Cells all share the same name space
- LibParts share the same name space
- Symbols of all types share the same name space
- GraphicObjects other than those above share the same name space

Thus, it is possible to have a LibPart, a Package, and a Cell named Fred and a Schematic View named Fred: they are each in a different name space. It is not possible to have a Global symbol and a Port symbol named X: they both reside in the same name space for symbols.

When it is first created, a LibObject exists only in memory. IsPersistent is true only if the object has been saved to disk. IsModified is true if the in-memory image differs from the persistent image on disk.

When a LibObject is used as a template for the creation of instances on a page, a copy of the object is made in the Library’s or Design’s cache. If the object is such a copy, IsCachedCopy is TRUE.

GraphicObject

```
ENTITY GraphicObject

SUBTYPE OF (LibObject);

SUPERTYPE OF (OptimizerSymbol, ParameterSymbol, Symbol);

          Color:    ColorT;

          Vectors:SET [0:?] OF Vectors;

          BoundingBox:Rectangle;

END_ENTITY
```

The GraphicObject is the basic graphical object managed by the Library. It consists of all the basic LibObject attributes as well as a bounding box, a color, and a set of primitive graphical entities or Vectors. It provides a template from which instances of the graphic can be created on a schematic page. It is reusable in that it acts as a template for many different instances in many different designs.

The GraphicObject inherits all the attributes of the LibObject and BaseObject. GraphicObjects always permit UserProps and DisplayProps.

All GraphicObjects are independent, persistent objects. They may be read and saved independently of other objects in the Library.

See DboGraph.h

Symbol

```
ENTITY Symbol

SUBTYPE OF (GraphicObject);
```

```
ABSTRACT SUPERTYPE OF (LibPart,
                        NetSymbol,
                        PSpiceABMSymbol,
                        PSpiceSourceSymbol,
                        SimulationDirectiveSymbol,
                        StimulusSymbol,
                        TitleBlockSymbol)
          Pins:     LIST [0:?] OF SymbolPin;
END_ENTITY;
```

The Symbol is the base class for LibPart, all the various NetSymbols (PortSymbols, Global Symbols, ...), and all the PSpice simulation symbols(StimulusSymbol, PspiceSourceSymbol, …). It is a subtype of GraphicsObject and as such consists of a set of graphic Vectors, a set of DisplayProps, and a set of UserProps. The vectors of the base class define the body of the object. The UserProps on the BaseObject entity are simple (Name, Value) pairs. They do not have any graphical representation on the Symbol. The DisplayProps define methods for displaying the values of the UserProps and DBProps graphically. Many of the User Props may not be actually displayed, so it is desirable to separate these two entities.

In addition to the basic attributes it inherits from GraphicObject, a Symbol contains a list of SymbolPins. The pins are ordered so they can be matched to the Instance pins and Package pins by position. Pins provide connect points for wires and carry signal information. Applications can access SymbolPins through the Symbol by:

1. pin name
2. pin (x,y) location
3. pin index in the pin list

A Symbol is an independent, persistent object. It may be read and saved independently of other objects in the Library.

Note: The TitleblockSymbol should actually be derived from GraphicObject. The reason for this derivation is an accident of history better left unexamined

See dbosmbl.h

Example of Creating a New Symbol in a Library

```
CRect rcBodyRect = m_DefaultBodyRect;

CString szSymbolName = dlg.GetName();
```

```
UINT nObjType = dlg.GetObjectType();
DboValue::PinTypeT nPinType;

switch (dlg.GetObjectType())// get the type of symbol to create
{
          case DboBaseObject::GLOBAL_SYMBOL:
               pDboSymbol = (DboSymbol*)(pDboLib->NewGlobalSymbol(szSymbolName,
     dboStatus));
               nPinType = DboValue::POWER;
               break;

          case DboBaseObject::PORT_SYMBOL:
               pDboSymbol = (DboSymbol*)(pDboLib->NewPortSymbol(szSymbolName,
     dboStatus));
               nPinType = DboValue::IN;
               break;

          case DboBaseObject::OFF_PAGE_SYMBOL:
               pDboSymbol = (DboSymbol*)(pDboLib->NewOffPageSymbol(szSymbolName,
     dboStatus));
                         nPinType = DboValue::IN;
                         break;

          case DboBaseObject::TITLEBLOCK_SYMBOL:
               rcBodyRect = m_DefaultTitleBlockRect;
               pDboSymbol = pDboLib->NewTitleBlockSymbol(szSymbolName,
     dboStatus);
               break;

          default:
               ASSERT(FALSE);
}

if (dboStatus == DBO_SUCCEEDED)
```

```
//-------------------------------------------------------------------
// Add a default pin to the net symbol (unless this is a title block!)
//-------------------------------------------------------------------
{
          if (nObjType != DboBaseObject::TITLEBLOCK_SYMBOL)
          {
                dboStatus = pDboSymbol->NewSymbolPinScalar(dboStatus,
                                                     CString(""),
                                                     nPinType,
                                                     CPoint(0,0),
                                                     CPoint(0,0),
                                                     FALSE) == NULL))
               if (dboStatus.Failed())
               {
                    ThrowDboException(dboStatus);
               }
          }

          //-------------------------------------------------------------------
     -
          // add default bounding box
          //-------------------------------------------------------------------
          dboStatus = pDboSymbol->SetBoundingBox(rcBodyRect);
          if (dboStatus.Failed())
          {
               ThrowDboException(dboStatus);
          }
}
else
{
          ThrowDboException(dboStatus);
}
```

```
// -----------------------------------------------------------------------
// Add the graphic Vectors to the symbol
// -----------------------------------------------------------------------
```

SymbolPin

```
TYPE PinTypeT
ENUMERATION OF (IN, OUT, IO, OC, PAS, HIZ, OE, POWER);
END_TYPE;



ENTITY SymbolPin
SUBTYPE OF (BaseObject);
ABSTRACT SUPERTYPE OF (SymbolPinScalar, SymbolPinBus)
          Owner:    Symbol;
          PinName:StringT;
          PinType:PinTypeT;
          StartPoint:Location;
          HotSpot:Location:
          IsLong    Boolean;
          IsClock:Boolean;
          IsDot:    Boolean;
          IsNetStyle:Boolean;
          LeftPointing:Boolean;
          IsRightPointing:Boolean;
          IsVisible:Boolean;
          IsGlobal:Boolean;
END_ENTITY;
```

The SymbolPin defines the graphical representation of an entry point for a signal into a Part. It consists of two end points: the StartPoint and the HotSpot. The HotSpot is the point to which Wires will connect. The instantiated PortInst corresponding to the SymbolPin may be Visible or not when the Symbol is instanced on the SchematicPage. The SymbolPin is an abstract entity which provides all the basic characteristics for the SymbolPinScalar and SymbolPinBus.

The PinType determines the electrical characteristics and the signal flow.

A set of UserProps is associated with the SymbolPin. The DisplayProps define the graphical display of the desired UserProps or DBProps. Note that the display of the PinName and PinNumber need not be added as DisplayProps on the SymbolPin. Their display on a PortInst is managed by the UI based on attributes of the PartInst and defaults of the Design.

See dbospin.h

SymbolPinScalar

```
ENTITY SymbolPinScalar

SUBTYPE OF (SymbolPin)

WHERE

          -- name is constrained to NOT be a valid bus name.

END_ENTITY;
```

The SymbolPinScalar has no attributes beyond those of the base class. A SymbolPinScalar will be displayed differently and hence must be distinguished from a bus.

See dbospin.h

SymbolPinBus

```
ENTITY SymbolPinBus

SUBTYPE OF (SymbolPin)

          Members:  LIST [0:?] OF StringT;
```

WHERE

- name constrained to have proper bus syntax of form <basename>[m..n]
- members can be accessed by bus syntax of form <basename>i where m<=i<=n

```
END_ENTITY;
```

The SymbolPinBus represents a list of pins. There is only a single graphic and property set; hence, only the names of the members are available (see dbobus.h). The SymbolPinBus is constrained to have a name with proper bus syntax, and the members of the bus are determined by the bus name.

See dbospin.h.

Example of Getting Member Names of PortBus

```
//--------------------------------------------------------------------------------
     -------
// Build up a map of all scalar members of this bus port
//--------------------------------------------------------------------------------
     -------
DboBusMemberNamesIter MembersIter(pPin->GetPinName(status));
while (!(PinName=MembersIter.NextMemberName(status)).IsEmpty()
{
          // Build up map of bus members
          PinBusMembers.SetAt((const char*)PinName, pPin);
}
```

LibPart

```
ENTITY LibPart
SUBTYPE OF (Symbol);
          Reference:StringT;
          PartValue:StringT;
          Contents: OPTIONAL View;
          PinNamesAreVisible:BooleanT;
          PinNamesAreRotated:BooleanT;
DERIVE
          ReferenceDesignator:StringT;
END_ENTITY;
```

LibParts in a Library define the graphical characteristics of a Part to be placed on a Schematic as well as certain electrical characteristics. Its graphical attributes are inherited from the base class GraphicObject and its electrical interface is inherited from the list of SymbolPins of the underlying Symbol.

The LibPart may optionally specify a Contents view. This View is normally a Schematic whose connectivity gives the electrical description of this LibPart. When the Part is instanced on a

Page, the Contents are copied to the instance and may be overridden at the instance level. The user may also add a Contents View to an instance whose original LibPart is primitive.

The Reference acts as a template for building the ReferenceDesignator for the resulting instances. The Reference will normally be a single character such as “U” or “R”. That is concatenated with “?” to compute the ReferenceDesignator, “U?”. This ReferenceDesignator is subsequently concatenated with the Device Designator to produce the default ReferenceDesignator for instances placed on the schematic, for example “U?A”. The Reference, PartValue, UserProps and DisplayProps of the LibPart are copied to the PlacedInst at the time it is placed.

Because PinNames are not displayed through DisplayProps, information about their display is held in the LibPart. PinNamesAreVisible and PinnamesAreRotated indicate to the UI how to draw the pins.

There may be several LibParts which are electrically equivalent. Such LibParts may be grouped into a Cell to indicate they are equivalent representations of the same electrical part. The SDT 386 part and its convert become two different LibParts grouped into a single Cell.

A LibPart is the graphical representation of a logical electrical entity. There may be more than one physical package placed on a printed circuit board that could physically embody the logical entity. The physical package is modeled by the Package entity. The Package contains a list of Devices just as a physical package could contain circuitry for several logical gates. Each Device identifies a Cell as the entity which can be packaged there. When the user places a part on the SchematicPage, he first selects a Package and a Device within that Package. He then identifies which Part in the associated Cell is to be placed. A PlacedInst always knows the Device it was created from in order that it can be packaged at a later time.

Currently there is no verification in the DB that all the LibParts in a Cell have anything in common such as pin out or name, nor is there any limit on the number of LibParts that may be placed in a Cell. *The UI, however, does impose a number of limitations on this relationship*. *A Cell may contain at most two lib parts and they must be named <CellName>.NORMAL and <CellName>.CONVERT.*. *More importantly, LibParts currently created through the UI must be associated with a Package. The Cell must also have the same name as the Package*. *This packaging requires that the pins of the LibParts correspond to the pins of the Package's Device*. *See the description of the Package object for more information on this correspondence*. *The DB does not enforce this relationship: the calling application is responsible for creating valid packages*. *In addition, the LibPart may not currently be saved independently of the Package: use the DboLib::SavePackageAll() function to save the Package, its Cells, and its LibParts in a consistent update.*

See dbolpart.h

Cell

```
ENTITY Cell
SUBTYPE OF (LibObject)
          Parts:    SET [0:?] OF LibPart;
END_ENTITY;
```

The Cell provides a mechanism for identifying those LibParts in the Library which the user feels are electrically equivalent. In some cases the difference may be only the graphical representation as in the case of the DeMorgan symbol. In other cases the difference may be in property values. For example, two parts might differ only in timing information or one part might carry properties appropriate for a military product and the another for a commercial product. The user chooses what he groups together in the Cell; there are no constraints applied by the system. If the user intends to go to PCB layout, all the Parts in a Cell should be "packageable" in the same device. Parts which cannot be packaged in the same Device must be in different Cells.

The Cell does not "own" its members. The LibParts are owned by the Library.

Note: The current release of Capture permits at most two LibParts in a Cell. They must have the same basename and have extensions: “.NORMAL” and “.CONVERT”.

Although the intention of the Cell is to group like LibParts, there is no restriction imposed on the grouping by the DB. Furthermore, the need to support translated parts requires that the Cell contain LibParts that have radically different pinouts that result from the use of “Pin Zero”. Such LibParts would normally be in different Cells with the current architecture and the Package would be “heterogeneous”.

See dbocell.h

Package

```
ENTITY Package
SUBTYPE OF LibObject
          Size:          Int;
          Devices:  LIST [0:?] OF Device;
          Designator:    StringT;
          PCBLib:   StringT;
          PCBFootprint:  StringT;
DERIVE
```

```
          IsHomogeneous:BooleanT;
END_ENTITY;
```

A Package models a PCB package. It provides the mapping of logical LibParts to physical packages. Many physical PCB packages contain more than one single electrical entity. For example, the 7400 contains 4 gates. The Devices of the Package correspond to the set of electrical entities that may be packaged within it. The list of Devices maps the pins of the logical entity (the LibPart) to the physical pins of the PCB package.

The PCBLib is the filename of the PCB library, and the PCBFootprint is the name of the PCB footprint for this package.

The Designator is a string such as "U?" which acts as a template used by the packager to assign the “Reference” for instances and occurrences in the Design. This will be combined with the Device's Designator to create a ReferenceDesignator on the PartInst or InstOccurrence.

The Devices contained within the Package are ordered and each references a Cell that defines the logical gate associated with this Device in the Package. If all the Devices reference the same Cell (the typical case), the Package is “homogeneous.” If there are different Cells referenced by the Package’s Devices, then the Package is “heterogeneous.”

A Package is an independent, persistent object. It may be read and saved independently of other objects in the Library. For reasons of performance, the current implementation reads and saves all associated LibParts and Cells along with the Package.

See dbopkg.h

PackageAlias

```
ENTITY PackageAlias
          PackageName:StringT;
          AliasName:StringT;
END_ENTITY;
```

Many packages have several names because they have different vendors, but their PCB characteristics are identical. Rather than create a separate Package (with its associated Cells and LibParts) in the Library, it is possible to associate different names with a single library Package. The user may lookup a Package by any of these aliases as well as the Package name.

Notice that the PackageAlias is not derived from BaseObject. It has none of the basic properties normally provided by that base class.

Note: In reality, these packages DO NOT have identical PCB characteristics. These should be different packages which all reference the same Cell and LibPart. The current "containment" relationship between the package and part make it impossible to provide different packages for the same part. In some future release the package, Cell, and LibPart relationships should be corrected. Note also that the effective props of an instance should look at the Package after the instance and before the LibPart to pick up any package specific properties. This should also apply to the effective props for all PortInsts.

See dbolib.h and dbopkg.h

Device

```
ENTITY Device

SUBTYPE OF (BaseObject)

          Owner:    Package;

          Cell:     Cell;

          Designator:         StringT;

          PinNumber:          LIST [0:?] OF StringT;

          PinInstantiated:    LIST[0:?] OF BOOL;

END_ENTITY;
```

The Device maps a set of pins in a physical PCB package to a logical Cell. The Designator is a unique identifier of the Device within the Package and is combined with the Designator of the Package to form the Reference Designator of the PartInst when a part is placed. The Designator is normally a single integer or Alpha character. The PinNumbers are the identifiers of the pins of the physical package and map one-to-one by position to the Pins of the LibPart.

The Pins of the LibParts contained within the specified Cell must correspond to the pins of the Device (and to the pins of the placed PartInst). This correspondence is maintained by the position of the pins within the pin lists of these objects. A SymbolPin at index i within the LibPart's list of pins maps to the PinNumber at index i within the Device. The table below shows the relationship of a Libpart with 4 pins (IN1, IN2, OUT, and POWER) with a Package of 10 pins: 3 Devices of 4 pins each plus 1 shared power pin.

Part	**Device A**		**Device B**		**Device C**	
Pin name	Number	instantiated	Number	instantiated	Number	instantiated
IN1	1	FALSE	7	FALSE	4	FALSE
IN2	2	FALSE	8	FALSE	6	FALSE
OUT	9	FALSE	3	FALSE	5	FALSE
POWER	10	TRUE	10	FALSE	10	FALSE

When multiple parts are packaged into a PCB package there are typically power and ground pins associated with the physical package that are shared among all the logical parts. These pins are normally global and it is, therefore, neither required or desirable to see them in the schematic. The IsVisible flag on the LibPart’s SymbolPin causes these pins to be invisible in the schematic. The user can override this visibility on an instance-by-instance basis by turning on the visibility of the pins for a particular instance. Once visible, the pin can be connected to a wire (thus isolating the global signal). Since the pin is shared, the user must connect all remaining pins associated with that same package/pinNumber to the same Net or the netlisters will issue an error. An alternative is to restrict the global pin’s instantiation to a single device within the package. For example, the user might wish to restrict the global pins to the ‘A’ device of a 4-device package. This is accomplished by setting the PinInstantiated flag to FALSE. When an part is placed, the PortInst is associated (by list position) with a PinNumber and PinInstantiated flag in the Device. If the PinInstantiated flag is FALSE, this pin will not be displayed on the instance, regardless of the pin visibility setting of the instance. The pin will not be extracted in the netlists or checked by DRC. The PortInst iterators will by default not return this PortInst. In the example above, instances of Device A will display 4 pins, whereas instances of Device B and C will display only 3.

This approach to sharing pins by not instantiating all the pins has the disadvantage that the Device with the single instantiated pin MUST be placed for each resulting package or there will be no global signal for the package in the netlist. If the user failed to place DeviceA of the example package above, there would be no net on pin 10 of the resulting netlist. The recommended method is to use the IsVisible flag on the LibPart.

See dbodevic.h

NetSymbol

```
ENTITY NetSymbol
SUBTYPE OF (Symbol);
ABSTRACT SUPERTYPE OF (OffPageConnectorSymbol,
                        PortSymbol,
                        GlobalSymbol,
                        ERCSymbol,
                        BookMarkSymbol)
```

WHERE

-- constrained that there is a single pin.

```
END_ENTITY;
```

NetSymbols in a Library define Symbols to be placed on a SchematicPage. When this Symbol is instanced on a Page, it adds additional characteristics to the Wire or PortInst to which it connects.

The NetSymbol differs from its supertype Symbol only by the constraint that there is only one pin. Such pins will not be treated as “pins” on nets in a netlist, but rather act as connect points to identify the object whose characteristics are to be altered.

Note: The ERCSymbol and BookMarkSymbols should actually be derived from GraphicObject. The reason for this derivation is an accident of history better left unexamined.

See dbonsmbl.h

OffPageSymbol

```
ENTITY OffPageSymbol
SUBTYPE OF (NetSymbol)
END_ENTITY;
```

The OffPageSymbol can be placed on a Page to create an instance of an OffPageConnector. Such an instance indicates electrical signals of the same name are to be connected between pages of the same Schematic.

See dbonsmbl.h

PortSymbol

```
ENTITY PortSymbol
SUBTYPE OF (NetSymbol)
          PinType:PinTypeT;
END_ENTITY;
```

The PortSymbol can be placed on a Page to create an instance of a Port. A Port indicates signals are to be connected vertically through the hierarchy from PortInst above to Port below.

See dbonsmbl.h

GlobalSymbol

```
ENTITY GlobalSymbol
SUBTYPE OF (NetSymbol)
END_ENTITY;
```

The GlobalSymbol can be placed on a Page to create an instance of a Global. A Global indicates that a signal is "global" in the Design. All global nets with the same name are considered connected anywhere in the design. Such signals are usually of the type POWER.

See dbonsmbl.h

BookMarkSymbol

```
ENTITY BookMarkSymbol
SUBTYPE OF (NetSymbol)
END_ENTITY;
```

The BookMarkSymbol can be placed on a Page to create an instance of a BookMark. The BookMark’s hot spot identifies a point on the Page that the user wishes to find by name.

See dbonsmbl.h

ERCSymbol

```
ENTITY GlobalSymbol

SUBTYPE OF (NetSymbol)

END_ENTITY;
```

The ERCSymbol can be placed on a Page to create an instance of an ERC marker. The ERC’s hot spot identifies an object on the Page where a design rule has been violated.

See dbonsmbl.h

TitleBlockSymbol

```
ENTITY TitleBlockSymbol

SUBTYPE OF (Symbol);

          PageCount:StringT;

          PageNumber:StringT;

          PageSize: StringT;

          CreateDate:StringT;

          ModifyDate:StringT;

END_ENTITY;
```

The TitleBlockSymbol should more properly be derived from GraphicObject since it has no pins. It is a reusable graphic saved in a Library which can be placed on a Page to create an instance of a TitleBlock. It carries identifying information about the page.

See dbotitle.h

ParameterSymbol

```
ENTITY ParameterSymbol

SUBTYPE OF (GraphicObject);

END_ENTITY;.
```

The ParameterSymbol inherits all the graphical characteristics of the GraphicObject. The ParameterSymbol is a reusable graphic saved in a Library and can be placed on a Page to create a ParameterInstance.

See DboPSmbl.h

OptimizerSymbol

```
ENTITY OptimizerSymbol

SUBTYPE OF (GraphicObject);

END_ENTITY;.
```

The OptimizerSymbol inherits all the graphical characteristics of the GraphicObject. The OptimizerSymbol is a reusable graphic saved in a Library and can be placed on a Page to create an OptimizerInstance.

See DboPSmbl.h

StimulusSymbol

```
ENTITY StimulusSymbol

SUBTYPE OF (Symbol);

          Stimulus: StringT;

END_ENTITY;
```

The StimulusSymbol is derived from Symbol. This provides it all the graphical characteristics of a GraphicObject and a list of pins. In addition to its inherited attributes, a StimulusSymbol has a single Stimulus attribute which holds the pathname of a stimulus file associated with the symbol. When the symbol is placed, the instance specific stimulus attribute defaults to the symbol’s value. It is a reusable symbol saved in a Library and can be placed on a Page to create a Stimulus.

See DboPSmbl.h

PSpiceSourceSymbol

```
ENTITY PSpiceSourceSymbol

SUBTYPE OF (Symbol);

END_ENTITY;
```

The PSpiceSourceSymbol is derived from Symbol. It is a reusable symbol saved in a Library and can be placed on a Page to create a PSpiceSource instance.

See DboPSmbl.h

PSpiceABMSymbol

```
ENTITY PSpiceABMSymbol

SUBTYPE OF (Symbol);

END_ENTITY;
```

The PSpiceABMSymbol is derived from Symbol. This provides it all the graphical characteristics of a GraphicObject and a list of pins. It is a reusable symbol saved in a Library and can be placed on a Page to create a PSpice ABM instance.

See DboPSmbl.h

PSpiceSimulationDirectiveSymbol

```
ENTITY PSpiceSimulationDirectiveSymbol

SUBTYPE OF (Symbol);

END_ENTITY;
```

The PSpiceSimulationDirectiveSymbol is derived from Symbol. This provides it all the graphical characteristics of a GraphicObject and a list of pins. It is a reusable symbol saved in a Library and can be placed on a Page to create a Stimulus.

See DboPSmbl.h

ExportBlock

```
ENTITY ExportBlock

SUBTYPE OF (Schematic);

END_ENTITY;
```

There are times when a user wishes to extract a set of connectivity from a SchematicPage and save it within a Library as a reusable entity. The ExportBlock permits storage of such reusable connectivity. It is a collection of connectivity entities whose owner is a Library. It is not part of a schematic, but rather is derived as a specialization of Schematic.

The assumption is that the objects added to an ExportBlock are being copied from an existing SchematicPage. Hence, the "create" functions act more as a copy from an original.

See dboxport.h.

View

```
ENTITY View

SUBTYPE OF (LibObject)

ABSTRACT SUPERTYPE OF (Schematic, ExternalView)

END_ENTITY;
```

A View describes the "contents" of an electrical device. It is an abstract base class that may be either a Schematic which describes the device graphically, or it may be an ExternalView which describes the device by identifying an external file as the container of the electrical description.

Note: *We are not currently using the ExternalView to support VHDL or PSpice views although this object was defined for such support.*

All Views are independent, persistent objects. They may be read and saved independently of other objects in the Library or Design.

See DboView.h

ExternalView

```
ENTITY ExternalView

SUBTYPE OF (View)

          ViewType:StringT;

          FileName:StringT;

END_ENTITY;
```

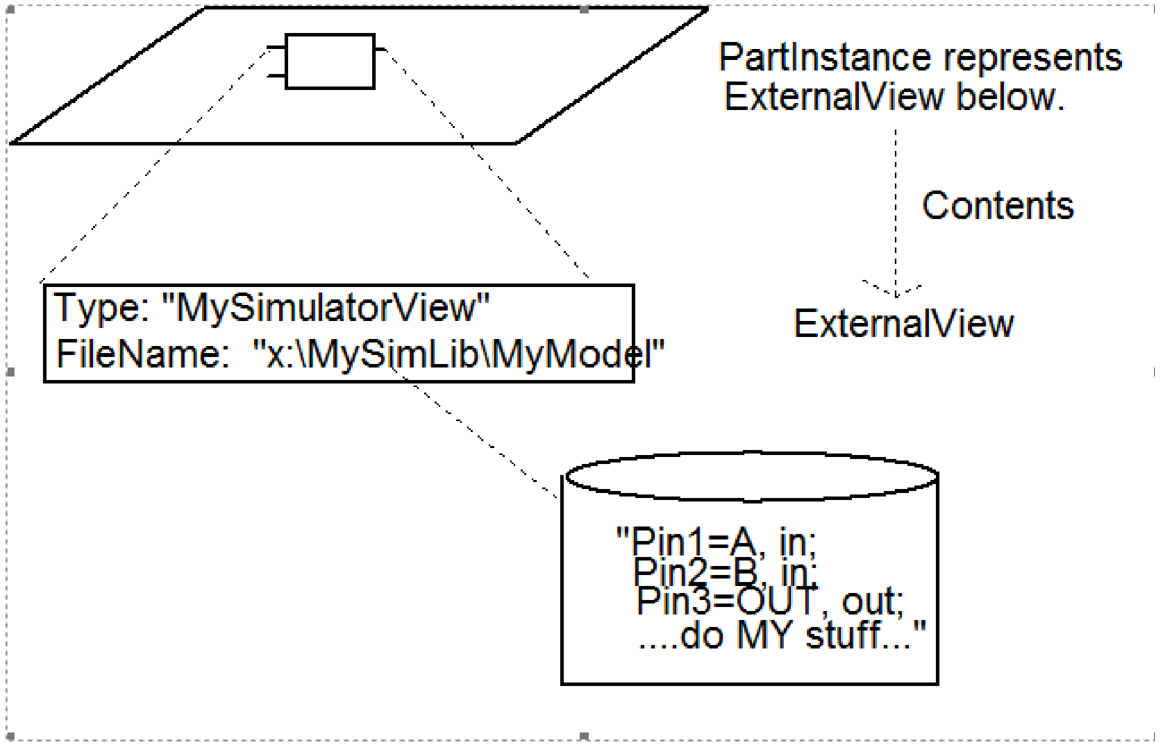
An ExternalView provides a mechanism for supporting an external description of an electrical object. An ExternalView defines the contents of a PartInst in an external file just as the Schematic provides the contents in graphics. The ExternalView consists of a ViewType string and a FileName string.

The semantics of the data contained within the file are not known to the Capture Schematic editor. It is known only to the external, downstream tool.

The ExternalView’s FileName specifies a file containing the description or “model” of the View. It may be a VHDL description, VendorX’s simulation model, an EDIF description, ... It may be ASCII text or it may be binary, whatever the format for that model may be. The ViewType is a string that identifies what that model may be used for. For a VHDL model, he Type would be set to "VHDL " and the FileName would contain a path to the ASCII VHDL description. If the user wanted to go to an MGC simulator, he might set the ViewType to “MGC” and the FileName would contain a path to the QuickSim binary simulation model.

A VHDL netlister would traverse the Design hierarchy, descend through a PartInst and get an ExternalView as the contents. It would examine the Type and determine that this was a "VHDL " description. The netlister would then read the VHDL description from the file specified by the FileName and add that description to the netlist.

A netlister for a different tool would follow the same procedure. Since that netlister did not understand VHDL, it would treat this PartInst as if it were primitive.



Other netlisters might simply include the FileName in the netlist and let the final tool access the data. This would be typical of a simulator with binary models.

In many ways ExternalViews are similar to UserProps. They both support tool specific information that the system stores without understanding the underlying semantics. They differ in several ways. UserProps are designed to support graphical display on the schematic and may be added and overridden through back annotations. The UserProp is owned directly by an object in the SchematicPage. ExternalViews are not displayable directly on the SchematicPage. The ExternalView behaves as any other contents View; the user would need to descend the hierarchy to see this information. An ExternalView may act as the contents

View of many PartInsts. It is a reusable object created within a Library or Design with the same sort of usage model as a Schematic.

A ExternalView is always contained in a Design or Library. It may be specified as the contents of a PartInst.

Note: The UI currently does not support ExternalViews through this mechanism.

See dboxview.h

NetGroup

```
TYPE BundleTypeT

ENUMERATION OF (BUNDLE, SCALAR, BUS, ILLEGAL);

END_TYPE;


ENTITY Bundle

SUBTYPE OF (BundleBusMember);

          mMembers: CMapStringToPtr;

          mMembersIndexed: CPtrArray;

          mOwner: DboBaseObject;

          m_NGSourceLibName: CString;

DERIVE

          mName: CString;

          mType: BundleTypeT;

          mBundleTemplateMap: BundleTemplateMap;

          mParents: CPtrArray;

          mMsb: Int;

          mLsb: Int;

END_ENTITY
```

The Netgroup is a bundle of nets derived from BundleBus Member. It can be classified as Scaler, Bus, Netgroup of Netgroups.

BundleTempleteMap is a generic container that contains all the Netgroup classes. All the netgroup-aware devices are referenced using the BundleTempleMap class.

Schematic Page Objects

Page

```
ENTITY Page

SUBTYPE OF (BaseObject);

          Name:          StringT;

          Owner:         Schematic;

          TimeCreated:   TimeT;

          TimeModified:  TimeT;

          PageNumber:    int;

          SizeName: StringT;

          TitleBlocks:   SET [0:?] OF TitleBlock;

          PartInsts:     SET[0:?] OF PartInst;

          Wires:         SET [0:?] OF Wire;

          BusEntries:    SET [0:?] OF BusEntry;

          Globals:  SET [0:?] OF Global;

          Ports:         SET [0:?] OF Port;

          OffPageConnectors:  SET [0:?} OF OffPageConnector;

          ERCs:          SET [0:?] OF ERC;

          BookMarks:     SET [0:?] OF BookMark;

          ParameterInstances: SET [0:?] OF ParameterInstance;

          OptimizerInstances:SET [0:?] OF OptimizererInstance;

          PSpiceABMInstances:SET [0:?] OF PSpiceABMInstance;

          PSpiceSimulationDirectiveInstances:SET [0:?] OF
     PSpiceSimulationDirectiveInstance;

          PSpiceSourceInstances:SET [0:?] OF PSpiceSourceInstance;

          StimulusInstances:SET [0:?] OF StimulusInstance;

          CommentGraphics:SET [0:?] of GraphicInstance;

          Width:         int:

          Height:        int:

          IsMetric: BooleanT;

          BorderDisplayed:BooleanT;
          BorderPrinted:BooleanT;
          TitleBlocksDisplayed:BooleanT;
          TitleBlocksPrinted:BooleanT;
          ANSIGridRefs:BooleanT;
          SizeName: StringT;
          OuterBorderLineStye:LineStyleT;
          OuterBorderWidth:LineWidthT;
          OuterBorderIsVisible:BooleanT;
          InnerBorderMargin:Location;
          InnerBorderLineStyle:LineStyleT;
          InnerBorderWidth:LineWidthT;
          InnerBorderIsVisible:BooleanT:
          BorderColor:ColorT;
          LabelFont:FontT;
          LabelColor:ColorT;
          HorizontalLabelCount:Int;
          HorizontalLabelWidth:Int;
          HorizontalLabelIsChar:BooleanT;
          HorizontalLabelIsVisible: BooleanT;
          HorizontalLabelAscending:BooleanT;
          VerticalLabelCount:Int;
          VerticallabelWidth:int;
          VerticalLabelIsChar:BooleanT;
          VerticalLabelIsVisible: BooleanT;
          VerticalLabelAscending:BooleanT;
          HorizontalSeparatorLineStyle:LineStyleT;
          HorizontalSeparatorWidth:LineWidthT;
          VerticalSeparatorLineStyle:LineStyleT;
          VerticalSeparatorWidth:LineWidthT;
          UserStorageIStorage;
DERIVE
          Nets:          SET [0:?] OF Net;
          Edits:         SET [0:?] OF BaseObject;
END_ENTITY;
```

A Page provides a description of a set of electrical connectivity. It is a graphical description made up of PartInsts, Wires, Ports, OffPageConnectors, and Globals. The Page also carries user definable borders, TitleBlocks and any amount of comment text and graphics the user wishes to place. Users of PSpice can also place instances of ParameterSymbols, OptimizerSymbols, StimulusSymbols, PspiceSourceSymbols, PSpiceSimulationDirectiveSymbols, and PSpiceABMSymbols. These instances provide information to the PSpice simulation.

The user normally builds this connectivity by “placing” LibParts, Symbols and Wires. The LibPart is not actually "placed" on the schematic in the sense that a physical part might be removed from a carton and plugged into a socket on a board. The Part is not moved nor is it exactly copied. Instead it acts as a template in the creation of a new type of object, a PartInst. Many of the attributes of the PartInst are determined by its LibPart, but it has characteristics of its own. A PartInst has a location on the Page, a color and a rotation. These are attributes of the PartInst that are not present on the LibPart. The user may add UserProps to the PartInst that do not exist on the original LibPart or may override the default UserProps of that Part. Although the Reference, Designator, and PartValue for the PartInst are seeded from the LibPart, the user may change these values interactively or with a variety of batch tools. Likewise, the DisplayProps of the PartInst are defined by the LibPart are merely the defaults. The user may change the display characteristics of these properties individually on the PartInst or even delete them.

The LibPart and Package data required for display of the PartInst will be saved locally in the Library or Design cache. This will make it possible to display the Page correctly even if the original Library is lost. The PartInst will retain knowledge of its original source and can be updated to correspond to the current part when requested.

Each PartInst presents a set of PortInsts around its border which correspond to the SymbolPins of the LibPart. Wires connecting these PortInsts are represented as polylines starting at one PortInst and connecting to another.

The user may draw the instance directly on the Page if he is working in a top down manner. Regardless of whether the instance is placed or drawn, the user may specify a Schematic or ExternalView which provides the Contents or lower level description of the electrical characteristics of this PartInst. If the original LibPart contained a default Contents specification, the user can override that specification on the PartInst. Through this mechanism, the user defines a hierarchy.

A Page is not simply a drawing; the entities it displays also carry electrical semantics. A Wire is not simply a line; the Page maintains knowledge of electrical connectivity at all times; connectivity is not computed only when the netlisters execute. The user may attach a set of

Aliases to the Wire. The Page recognizes that Wires with the same Alias are electrically connected. It similarly understands that wires that cross are not electrically connected unless a point is placed. This knowledge permits the Page to display the Net and its properties as a whole rather than simply displaying lines that represent Wires. It also permits a higher level of early error detection.

Ports connect wires on one level of the hierarchy with nets at the next higher level. Each Port at this level corresponds to a PortInst of the PartInst above. A Wire that connects to the PortInst above is "connected" to the Wire connected to the Port below. OffPageConnectors "connect" Wires on one Page to Wires with the same Name or Alias on other Pages in the same Schematic. Globals may be connected to Wires to indicate which Wires represent signals that are “global” to the design. Such signals connect across page and schematic borders by name. Ports or OffpageConnectors are not required. The most typical use of Globals is to create power and ground signals through the design.

Edits on a page may be grouped into transactions. A transaction is a set of edits to the page enclosed with Start and Commit Transaction requests to the page.

```
pPage->StartTransaction();

// perform edits

pPage->CommitTransaction();
```

If the user wishes to UNDO the set of edits, he must call AbortTransaction rather than Commit. This will remove any edits from the database since the last StartTransaction().

```
pPage->StartTransaction();

// perform edits

pPage->AbortTransaction();
```

During interactive editing, a transaction encloses a single command, but that command may make many edits to the database (e.g.: a block move). At the start of a command the application determines if a transaction is active. If so, it calls CommitTransaction() to terminate the previous command and then calls StartTransaction() to begin the next. If the next command is UNDO, the application calls AbortTransaction() rather than Commit. If the next command is REDO, the application calls AbortTransaction() to undo the previous UNDO command.

Batch processes normally would not wish to use the transaction mechanism because of the overhead of maintaining the log of such a large number of edits.

The Nets are maintained continuously; they are not just computed as part of the netlist. Net computations are limited to the page. They do not consider connectivity from other pages of the Schematic or other Schematics in the Design. This limitation is due to space and performance considerations. The SchematicNets (schematic-wide signals) and FlatNets (design-wide signals) are computed only when needed during display of the design in physical mode, netlisting or design rule checking.

Net computations are not performed for each update to the DB. Rather, they are computed after a set of edits. During interactive editing the application will normally compute the connections after each command in order that the connect information be displayed before subsequent edits. Thus, an interactive application should call Connect() just before committing the transaction to compute connections for the objects edited in the last transaction. A batch process would normally call Connect(TRUE) at the end of its processing to compute connections for the entire page.

As a result of Net computations, edits to one object on the page may alter information displayed on other objects. For example, editing a Wire by moving its endpoint to connect to a PortInst alters the PortInst. Its Wire and Net are no longer NULL. The application may wish to show this visually by removing the not-connected dot on the PortInst. Even more important to the application, objects may be added or deleted from the database as a result of the edit. For example, moving a Wire’s endpoint so that it connects to the endpoint of another Wire segment results in the 2 Wire’s being merged into a single Wire. If the application has a pointer to both Wires, it must “learn” that one of them has been deleted and delete its pointer. It is also important to reduce the amount of redrawing the application must do: it should not need to extract the current state of the DB and redraw the entire page after each edit. The DB Page, therefore, maintains a list of objects that have been modified during a transaction. The application may iterate over these objects to determine which have been added or deleted and which have changed with the DboPageEditsIter.

A UserStorage area is provided for the use of applications to store additional data which is associated with the page but not directly managed by the DB. Similar UserStorage areas are available for the Library and the Schematic. The data within the storage is not available unless that application is running; standard Capture does not “understand” it. It is essentially a very large UserProp attached to the page.

See dbopage.h

Example of Start and End Transaction

```
     //************* Perform edits on all the pages of this schematic
*************

     DboSchematicPagesIter* pPagesIter = NewPagesIter(status);

     DboPage* pPage = NULL;

     while ((pPage=pPagesIter->NextPage(status)) != NULL) {

          // *********start the transaction *********
```

```
        pPage->StartTransaction();

        // ***************************************************
        // ******* Update the Objects on the page *******
        // ***************************************************

        // ***************************************
        // Cleanup Wires and Connect Nets
        //***************************************
        status = pPage->Connect();

        // **********************************
        // Check for errors during Connect
        // Rollback changes if failed
        //**********************************
        if (status.Failed()) {
             pPage->AbortTransaction();
             delete pPagesIter;
             return status;

        // ************************************
        // Commit the Changes
        // ************************************
        pPage->CommitTransaction();

        }
        //**************************************************
        // make sure the page gets saved
        //**************************************************
        SavePage(pPage);
   }
   delete pPagesIter;
```

Example of Iterating over Edits to Page

All objects contained in the Page have a UserData field that is set by any application using this method of examining changes. The UserData should hold a pointer to an associated graphic object in the application or its ID. A NULL in the UserData field indicates that the object has been newly constructed and that no application object has yet been created.

```
//-------------------------------------------------------------------------------
     -----------------------------------

//  Update the display of all objects which were changed by the last command

//-------------------------------------------------------------------------------
     ----------------------------------

// Create in iterator over the edits to the Page

DboSchematic *pDboSchematic = (DboSchematic *)pDboPage->GetOwner(dboStatus);

DboPageEditsIter EditsIter(pDboPage);

DboBaseObject *pBaseObj;

void *pUserData;


// ***** Get the next edited object*****

while(EditsIter.NextEdit(pBaseObj, pUserData) == DBO_SUCCEEDED)

{

          // ***** DBO Object returned => edit existing or add new object*****

          if(pBaseObj != NULL)

          {

               // ***** Switch based on type of object *****

               int nType = pBaseObj->GetObjectType(dboStatus);

               switch(nType)

               {

                    case DboBaseObject::WIRE_SCALAR:

                    case DboBaseObject::WIRE:

                    case DboBaseObject::WIRE_BUS:

                    {
```

```
                    DboWire *pDboWire = (DboWire *)pBaseObj;// current DB
state
                    COrWire *pWire = (COrWire *)pUserData;// current UI state

                    //*********************************************
                    // UserData exists -> this is an update to an existing
wire
                    //   redraw the Wire to match its current DB state
                    //*******************************************
                    if(pWire != NULL)
                    {
                         rect = pWire->GetRect();
                         UpdateRectInViews(&rect, TRUE, FALSE);
InvalidateViewsRect(pWire->GetRect());
                         pWire->DbSync(pDboWire, this);
                    }
                    //**********************************************
                    // NO UserDate->new Wire
                    //   Create new UI entity for it
                    //**********************************************
                    else
                    {
                         // New Wire is Bus ->Create new COrBus
                         if(nType == DboBaseObject::WIRE_BUS)
                         {
                              pWire = new COrBus(*pPoint, pDboWire, ....
                         }
                         else
                         // New Wire is Scalar -> Create new COrWire
                         {
                              pWire = new COrWire(*pPoint, pDboWire, .....
                         }
```

```
                              // Add new Shape to graphic display
                              Add(pWire, FALSE, FALSE);

                              // Make the new Shape be selected
                              SelectShape(pWire, SHAPE__ONSHAPE, FALSE, FALSE);

                         }

                         //**************************************************
                         //   Update the Views
                         //***************************************************
                         // determine the bounding rectangle of the new shape
                          rect = pWire->GetRect();
                         // redraw the altered rectangle
                         UpdateRectInViews(&rect, TRUE, FALSE);
                         InvalidateViewsRect(pWire->GetRect());

                    //------- handle other types of objects --------
          }

          //*******************************************************************
          // No DB object exists -> must be Delete
          //*******************************************************************
          else
          {
               // ************ Get the shape fro the obj that has been deleted from
     DB ********
               COrShape *pShape = (COrShape *)pUserData;
               rect = pShape->GetRect();
               UpdateRectInViews(&rect, TRUE, FALSE);
               delete pShape;
          }
}
```

GraphicInstance

```
ENTITY GraphicInstance
SUBTYPE OF (BaseObject)
SUPERTYPE OF (PartInst,
               GraphicBoxInst,
               GraphicLineInst,
               GraphicArcInst,
               GraphicEllipseInst,
               GraphicPolygonInst,
               GraphicPolylineInst,
               GraphicCommentTextInst,
               GraphicBitMapInst,
               GraphicSymbolVectorInst,
               SymbolInstance,
               TitleBlock)
          Name:          StringT;
          Owner:         SchematicPage;
          SourceDefinition:GraphicObject;
          Location: Location;
          Rotation  RotationT;
          Mirror:        Boolean;
          BoundingBox:RectangleT;
DERIVE
          OffsetBoundingBox:RectangleT;
END_ENTITY;
```

The GraphicInstance represents the instantiation of a GraphicObject, Symbol or LibPart on a schematic Page. It specifies the Location, BoundingBox, Mirror and Rotation of the instance on the page. The BoundingBox is set by the application and the Location and Rotation are applied to that rectangle to provide the computed OffsetBoundingBox.

Note: *Although it is possible to create a GraphicInstance in the database, the current release of Capture cannot create or display such an object*. *Always create one of the specializations*.

See dbographi.h

SymbolInstance

```
ENTITY PartInst
SUBTYPE OF (GraphicInstance);
ABSTRACT SUPERTYPE OF (NetSymbolInstance,
                         OptimizerInstance,
                         ParameterInstance,
                         PartInst,
                         PSpiceABMInstance,
                         PSpiceSimulationDirectiveInstance,
                         PSpiceSourceInstance,
                         StimulusInstance);
          Pins:          LIST [0:?] OF PortInst;
DERIVE
          PinCount: Int;
END_ENTITY;
```

A SymbolInstance represents the instantiation of a symbol within a Schematic Page. It is a graphical entity which corresponds to some netlisted attribute of the page. In the case of PartInsts, the entity represents an electrical object in the design. NetSymbolInstances are graphical representations of attributes of the net or signal. They give instructions on how to connect nets in the design. The various PSpice instances give instructions to the PSpice simulator.

PartInst

```
ENTITY PartInst
SUBTYPE OF (SymbolInstance);
ABSTRACT SUPERTYPE OF (PlacedInst, DrawnInst);
          IsPrimitive:BooleanT;
          BoundingBox:Rectangle;
          Reference:StringT;
          PartValue:StringT;
          Contents: OPTIONAL View;
          PowerPinsVisible:BooleanT;
DERIVE
          DefiningPart:LibPart;
END_ENTITY;
```

A PartInst represents the instantiation of a primitive Part or a View within a Schematic. It is a graphical entity within a Page which corresponds to a connectivity entity, an instance, within the resulting netlist. The instance's characteristics may be determined from a LibPart. For a PlacedInst, the defining part is from a Library. A DrawnInst also has a defining LibPart, but that LibPart is created specifically for just this one instance when the instance is constructed. The application then adds the graphical Vectors to that LibPart after the instance has been constructed.

The list of PortInsts represent connection points. In a hierarchical instance, they correspond to the Ports of the Contents View. The PortInsts are ordered; each has a unique position within the instance and may be accessed by position as well as name. PortInsts may be buses or scalar. The PinCount is computed from the size of the list. PortInsts correspond 1-to-1 with the SymbolPins of the defining LibPart.

PinNames are displayed according to the PinNamesAreVisible and PinNamesAreRotated flags on the defining LibPart. Global SymbolPins are often marked as invisible on the LibPart. In some cases the user may wish to isolate the global signal for that pin by connecting it directly to a Wire, Port, or OffPageConnector. To connect to a PortInst, it must be visible. The PowerPinsVisible flag on the PartInst is initially set to FALSE, but may be set by the user to TRUE. When TRUE, the pin is by the UI and the point becomes an active connection point in the DB.

Note: Visibility does not alter its globalness; only connection to a wire or pin can change the normal global net behavior.

If a PartInst is hierarchical, it must have a Contents. The Contents is a Schematic or an ExternalView which contains a textual definition of the electrical characteristics of the instance. The Pins should correspond to the Ports of the Schematic below. A PartInst with no Contents is primitive in any Design that contains it.

A PartInst may have a hierarchical Contents, but the designer wishes it to be treated as primitive. If the IsPrimitive attribute is TRUE the netlister will not descend into the Contents.

Packaging information such as reference designator and Package Name are attached to the PartInst when it is placed (the user has selected a package/device for placement). These may then be edited interactively or as a result of such batch operations as Annotate.

The EffectiveProps of a PartInst are the result of “layering” the instance’s UserProps over those of the defining LibPart:

1. Return all the UserProps on the PartInst
2. Return those DBProps on the PartInst that do not conflict with UserProps already seen (the instance specific UserProps override the instance’s DBProps.)
3. Return those UserProps on the defining LibPart that do not conflict with any instance specific UserProps or DBProps already seen. (the instance specific properties override the part’s properties).

See dboparti.h

PlacedInst

```
ENTITY PlacedInst

SUBTYPE OF (PartInst);

          SourcePartName:StringT;

          SourcePackageName:StringT;

          SourceDeviceDesignator:StringT;

          SourceLibName:StringT;

          DeviceDesignator:StringT;

          PCBLib:        StringT;

          PCBFootprint:  StringT;

DERIVE

          SourcePart:    LibPart;

          SourcePackage:Package;

          SourceDevice:  Device;

          SourceLib:     Library;

END_ENTITY;
```

A PlacedInst represents an instance of a LibPart within a schematic Page. It also identifies a Package and a Device to be used when the instance is packaged. Perhaps a better name would be a “PackagedInst” since all PlacedInsts carry along information about their packaging.

A PlacedInst may be hierarchical or primitive. The usual default is primitive for designs destined for PCB layout, but will frequently be hierarchical for simulation designs.

The PortInsts of a PlacedInst not only correspond to the SymbolPins on the LibPart but also to the pins on the Device. The PinName is extracted from the LibPart and the PinNumber is extracted from the Device by indexing into these lists with the PortInst’s position. Some Devices have pins which are marked with IsInstantiated set to FALSE. Iterators over the PortInsts of such instances will by default skip the non-instantiated pins. (Applications that need access to such pins can set a flag on the iterator to force it to return all PortInsts). Non-instantiated PortInsts will not be displayed in the UI nor will they be extracted in the netlists.

Display of PinName and PinNumber in the UI are controlled by PinNamesVisible and PinNamesRotated attributes of the PortInst and the PartInst. The value displayed for the PinNumber is extracted from the Device. The value displayed for the PinName is extracted from the LibPart.

Because much of the LibPart, Package and Device information is cached within the Library, it is necessary to permit the user to "update" the PlacedInstance on demand to force it to conform to any changes in the original LibPart, Package or Device. All instances of the LibPart in the Design/Library must be updated to conform to the version of the part in the cache. Use the UpdateCache() function on the Library to bring a Design or Library up to date with a LibPart. In some cases the user needs to move a library or may wish to change to some library which contains alternate representations of the same set of parts. In such cases the user should call ReplaceCache() on the containing library.

The UI checks when a new PlacedInst is placed that the cached LibPart and Package are up to date with the original by calling the Library function CacheIsOutOfDate(). This checks the timestamps on the source part with the timestamp on the cached part and returns TRUE if the cache is older than the library version.

See dboplaced.h

DrawnInst

```
ENTITY DrawnInst

SUBTYPE OF (PartInst);

END_ENTITY;
```

Interactively a HierarchicalBlock is created directly on the schematic Page. The DB object corresponding to a HierarchicalBlock is a DrawnInst. There is no LibPart in any Library that is “placed” when a DrawnInst is constructed; a LibPart is constructed for the instance when the instance is constructed. The LibPart is created locally in the owning Library or Design and can be accessed only through the instance: the LibPart cannot be seen through iterations over the LibParts in the Library. There is no packaging information associated with such an instance. Graphic Vectors are added to this private LibPart not to the instance. Adding pins to the DrawnInst actually results in the creation of SymbolPins in the underlying LibPart.

The term “Hierarchical Block” is somewhat misleading since the block may be primitive. To be hierarchical the primitive flag must be set to FALSE (or DEFAULT and the design level default be set to hierarchical) and a valid Contents View must be specified.

The graphics need not be limited to a rectangle. Any Vectors may be added to the LibPart.

Note: The UI currently assumes the block is a simple rectangle defined by its bounding box and will not be able to display any additional graphics.

See dbodrawn.h

PortInst

```
ENTITY PortInst
SUBTYPE OF (BaseObject);
ABSTRACT SUPERTYPE OF (PortInstScalar, PortInstBus);
          Owner:    PartInst;
          IsNoConnect:Boolean;
          Wire:     OPTIONAL Wire;
          Position:int;
DERIVE:
          Net:      OPTIONAL Net;
          OffsetHotSpot:location;
          Name:     StringT;
          Number:StringT;
          SymbolPin:SymbolPin;
          IsGlobal:Boolean;
          PinType:PinTypeT;
          IsClock:BooleanT;
          IsDot:    BooleanT;
          IsLong:   BooleanT;
          IsNetStyle:BooleanT;
          IsRightPointing:BooleanT;
          IsLeftPointing:BooleanT;
          IsVisible:BooleanT;
END_ENTITY;
```

Every PortInst is owned by a PartInst. Its position determines which SymbolPin on the instance's defining LibPart was used in its construction. (The PortInsts of the PartInst are ordered in the same order as the SymbolPins of the defining LibPart.) The OffsetHotSpot is the location of the HotSpot defined by the SymbolPin after location, rotation, and mirror have been applied to the instance. The PortInst may not be visible if the defining SymbolPin is specified as not visible or if the associated PinNumber is set to non-instantiated; power and ground pins are frequently not visible on the PartInst. The OffsetHotSpot of the SymbolPin identifies the point at which a Wire may be connected.

PortInsts are typically created when an instance is placed and cannot be modified on a PlacedInstance. On a DrawnInst, pins may be added after the construction of the instance, but the act of adding them creates a defining SymbolPin on the instance’s underlying LibPart.

The PortInst Name is that of the defining SymbolPin. The PortInst Number is extracted from the Device’s PinNumber at the same position as the PortInst. It is possible to “swap” pins on the PartInst. On a DrawnInst this is simply provided by changing the name of the SymbolPin. Since the LibPart of a PlacedInst is shared, it is not possible to change the name so easily. The change is made by overriding the PinName and PinNumber of the PortInst through addition of UserProps with ID’s IDS_PROP_PIN_NAME and IDS_PROP_PIN_NUMBER. That should be generally invisible to the application if it uses EffectiveProps to access these attributes.

If a PortInst connects physically to a Wire, it is added to the PortInsts on that Wire’s Net. If a PortInst connects pin-to-pin to a Port, Global, or OffPageConnector, the PortInst is added to the Net associated with that object. If two PortInsts connect pin-to-pin, a Net is created to contain those two PortInsts. If a PortInst is global, it is added to a Net with the same name as the PortInst; a new Net is created if it doesn’t exist already. A PortInst need not be connected to anything; Wire and Net are both optional.

During interactive editing a small no-connect dot is displayed on all PortInsts that are not connected to a Wire or otherwise connected to a Net. DRC also issues a warning for unconnected pins. In some cases the user may wish to leave a PortInst dangling. In such cases, he may set the IsNoConnect to TRUE; the no-connect dot will not display nor will DRC issue a warning.

The set of DisplayProps permits the display of any additional properties which are not specified on the original Part's SymbolPins. Because of the memory and disk space required for DisplayProps, PinName display is controlled by attributes on the PartInst.

See dboporti.h

PortInstScalar

ENTITY PortInstScalar

SUBTYPE OF (PortInst)

SUPERTYPE OF (PortInstBusMember)

WHERE

-- PinName must NOT be a valid bus name

END_ENTITY;

The PortInstScalar has no attributes beyond those of the base class. It exists as a separate type to distinguish it from buses.

The PortInstScalar cannot connect to any bus objects (WireBus, PortInstBus, NetSymbolInst with Bus name). Although the hot spot may physically intersect the WireBus or coincide with the hot spot of a bus pin, no “electrical” connectin is made; the PortInst will not be added to the net that contains the bus object and the not-connected box will continue to display in the UI.

PortInstBus

```
ENTITY PortInstBus

SUBTYPE OF (PortInst)

DERIVE

          Members:  SET [0:?] OF PortInstBusMember;

WHERE

          -- PortInstanceBus must have a name and the name must have form
     <basename>[m..n]

END_ENTITY;
```

A PortInstBus represents a bundling of a list of PortInsts into a single entity. It differs from a PortInstScalar when it is displayed (the line is wider) and it will be treated differently within the netlist. The Members are owned by the bus in the sense that they are deleted when the bus is deleted, they cannot be members of more than one bus, and they cannot be moved to another bus or out of the bus.

A PortInstBus must have a Name which conforms to proper Bus syntax:

```
          <basename>[m..n]
```

The Members of the PortInstBus are computed from this name and have names of the form

```
          <basename><index> where index lies in the range m..n.
```

When a Wire is terminated at the HotSpot of a PortInstBus, the system checks if the Wire is also a bus. If not, no signals are connected. If the Wire is a bus, connections are built in the underlying schematic connectivity. The members of the NetBus corresponding to the WireBus are attached by position to the members of the PortInstBus.

See dboporti.h

PortInstBusMember

```
ENTITY PortInstScalar

SUBTYPE OF (PortInst)

DERIVE

          ContainingBus:PortInstBus;

WHERE

          -- PinName must NOT be a valid bus name

END_ENTITY;
```

The PortInstBusMember is a computed object. It is not returned when iterating over the PortInsts of the PartInst; only when iterating over the members of a PortInstBus.

The PinName is computed from the basename of the bus concatenated with the position of the member in the bus range. If the ContainingBus is on a NetBus, the PortInstBusMember is connected to the NetScalar at the same position in the NetBus' range. For example, if PortInst P[0..1] connects to Wire B[2..3] then PortInstBusMember P0 is on net B2 and PortInstBusMember P1 is on net B3.

All other attributes of the bus member are computed from its containing bus. Its position is that of the bus in the PortInst list. Its PinNumber is that of the ContainingBus. Its OffsetHotSpot is that of the ContainingBus. Its UserProps are those of the ContainingBus, ...

See dboporti.h

Wire

```
ENTITY Wire
SUBTYPE OF (BaseObject)
ABSTRACT SUPERTYPE OF (WireBus, WireScalar);
          Identifier:Int;
          Owner:    Page;
          Points:   SET[2:?] OF LocationT;
          Color:    ColorT;
          Aliases:  SET [0:?] OF CommentText;
DERIVE:
          Net:           Net;
          Name:          StringT;
          Junctions:     SET [0:?] OF LocationT;
          ConnectedWires:     SET [0:?] OF Wire;
          PortInstances:SET [0:?] OF PortInst;
          Globals:  SET [0:?] OF Global;
          Ports:    SET [0:?] OF Port;
          OffPageConnectors:  SET [0:?} OF OffPageConnector;
          BusEntries:    SET [0:?] OF BusEntry;
          BoundingBox:   Rectangle;
END_ENTITY;
```

The Wire is an abstract entity that may be either a WireScalar or a WireBus. It is a single segment, consisting of a start and an end point where it can physically connect to other Wires, PortInsts, Ports, OffPageConnectors, or Globals. It is the graphical representation of an electrical signal, a Net.

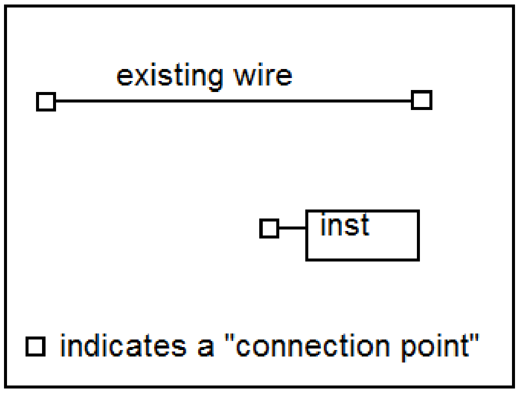
A Wire may be associated with any number of names. These names are added to the Wire through the creation of an Alias. Two Wires that have the same Alias name are electrically connected; they are associated with the same Net. A Wire that has an Alias with the same name as a Port, Global, or OffPageConnector is electrically connected to that object; they are on the same Net.

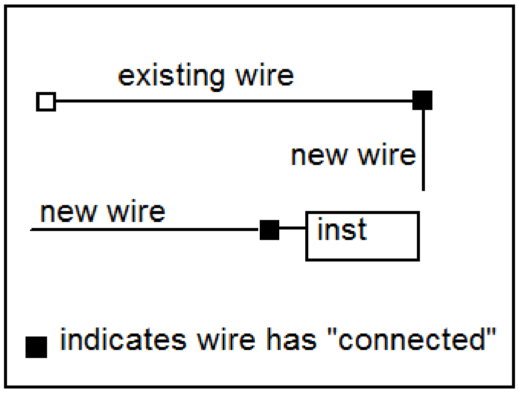
Every Wire is associated with exactly one Net which represents all the Wires, Ports, Globals, OffPageConnectors, and PortInsts connected either by physical connection or Alias.

When the user places a Wire he puts down 2 points. Each point of the Wire is a "connection point" as are the hot spots of the PortInsts, Ports, etc.

A connection is made if:

- a point of the newly placed Wire coincides with an existing connection point = an existing point of another Wire or hot spot ...





- The hot spot of a newly placed instance PortInst, Port, Global, or OffPageConnector is the endpoint of an existing wire.

See descriptions of Net objects for more information about the connecting Wires.

Junctions are only displayed at a "connection point," but not all connection points are Junctions

See dbowire.h

WireScalar

```
ENTITY WireScalar
```

```
SUBTYPE OF (Wire)
```

WHERE

-- Any alias attached to a WireScalar must NOT have a valid bus name.

```
END_ENTITY;
```

A WireScalar represents a single bit signal. A WireScalar may not connect to any bus.

See dbowire.h

WireBus

```
ENTITY WireBus

SUBTYPE OF (Wire)

DERIVE

          Members:  SET [0:?] OF Wire;
```

WHERE

-- WireBus must have a name and the name must have form <basename>[m..n].

```
END_ENTITY;
```

A WireBus represents a bundling of its Members into a single entity. It is displayed with a wider line and will be treated differently within the netlist.

A WireBus must have a Name which conforms to proper Bus syntax: <basename>[m..n]. Its Members are computed from the Name and have names of the form <basename><index> where index is in the range [m..n].

Buses are placed just as any other wire. Connect points are created following the rules described above.

A Wire is really a container for a set of scalar Nets rather than representing a single Net. When a WireBus is placed it is basically undefined until an alias is attached to it; it is the name which defines the nets that the Wire represents. Hence, a WireBus cannot really be "connected" unless it is named somehow. A WireBus (or any Wire) gets its names from several sources. I will call these the name sources:

- An Alias attached to the bus itself. Aliases that do not conform to valid bus syntax can not be attached to the WireBus. Valid syntax is the form <basename>[Lsb..Msb].
- A Port which has a valid bus name and is physically connected to the WireBus. Ports whose names are not valid bus names do not influence the net connections (other than they would if they were not physically connected).
- A Global which has a valid bus name and is physically connected to the bus
- An Off Page Connector which has a valid bus name and is physically connected to the bus

A bus may have numerous name sources and these all influence the final definition of the bus and its nets. When a name source is added to a WireBus through any of the above sources, the nets it defines are created by extracting the basename, Lsb and Msb from the source. Net scalars are created of the form:

```
    <basename><Lsb>
    <basename><Lsb>+1
        ...
   <basename><Msb>
```

{Note Msb may actually be smaller than Lsb, in which case the above is <Lsb>-1}

If there are other name sources attached to the bus, these newly created net scalars must be connected with the existing nets created by the existing aliases. The general rule is:

> **If one alias when expanded contains all the members of another, the smaller is simply "absorbed" into the larger. The members are all connected simply by name. If the basenames differ or the bus ranges are not completely contained within one another, the aliases are expanded into their individual scalar members and bitwise connected Lsb to Lsb, ..., Msb to Msb.**

The following examples may help make this rule clearer:

- Case 1: NewAlias basename is different than oldAlias basename: The net scalars are connected in a normal bitwise fashion from Lsb to Msb.

  Alias A[0..1] exists on bus. This results in the existence of nets A0, A1.

  Add Alias B[0..2]. This expands to B0, B1, B2.

  Bitwise connection results in connecting B0=A0, B1=A1. B2 is left unconnectedto any bits or A[0..1] since the range of A is smaller than that of B.

  Following standard net naming rules the resulting nets are: NetBus B[0..2] with members A0, A1, B2.

- Case 2: New Alias basename is the same as old Alias basename, but the new range contains the old. The new, larger bus is expanded and basically absorbs the smaller.

  Alias A[0..1] exists on bus. => A0, A1 expanded.

  Alias A[0..2] added to bus. => A0, A1, A2 expanded.

  A0 and A1 combine with the originals.

  The result is: NetBus A[0..2] with members A0, A1, A2.

- Case 3: New Alias basename is same as old, but new range is completely contained in the old. The old, larger bus absorbs the new smaller one.

- Case 4: New Alias basename is same as old but new range is disjoint. The two ranges are bitwise connected.

  Alias A[0..1] exists on bus => A0, A1 expanded

  Alias A[3..5] added to bus => A3, A4, and A5 expanded.

  A0 and A3 connect into single net named A0.

  A1 and A4 connect into single net named A1.

  The result is NetBus A[3..5] (wider bus range takes precedence) has a second alias A[0..1] and members A0, A1, A5. Net A0 has alias A3 and Net A1 has Alias A4.

- Case 5: Disjoint ranges exist on bus and containing bus range added. The containing range "absorbs" both the original buses.

  In the previous example, if the user subsequently adds a "master" bus which contains these 2 ranges, the more normal "absorb" rule takes over. In this example, if the user added a 3rd Alias A[0..5], the resulting netlist would be NetBus A[0..5] with 2 aliases A[0..1] and A[3..5] and 6 members A0, A1, …., A5.

- Case 6: New Alias basename is same as old but new range overlaps. The two ranges are bitwise connected.

  Alias A[0..2] exists on bus => A0, A1, A2 expanded

  Alias A[1..3] added to bus => A1, A2, A3 expanded

  Since neither range includes the other, the 2 ranges are bitwise connected => A0=A1, A1=A2, A2=A3

  The result is NetBus A[0..2] with alias A[1..3] and only one distinct scalar A0 with aliases A1, A2, A3. The 3 members of A[0..2] are all A0.

- Case 7: Aliases with different basenames on net and new alias added with same basename and containing range

  Alias A[0..1] exists on bus => A0, A1 expanded

  Alias B[2..3] exists on bus => B2, B3 expanded

  Bitwise connections connect A0=B2 and A1=B3.

  Alias B[0..7] added. This "absorbs" B[2..3} and bitwise connects with A[0..1] resulting in a different connection for the original scalar members. The original connection between A0 and B2 is now replaced by the bitwise connection of A0 *and B0. Similarly A1 reconnects to B1. The final netlist is Net Bus B[0..7] with 2 aliases A[0..1] and B[2..3]. The Bus members are A0, A1, B2, B3, ..., B7.*

In all the above examples the source of the name may either be an Alias on the Bus or any of the other name sources: physical connection with a Port, Global, or OffPageConnector.

Aliases can also be added by physically connecting one wire bus with another.

A bus has alias A[0..1] and a second had alias B[2..3]. The second bus connects physically with the first. Because the 2 wire Buses connect they define a single NetBus and the aliases on both wires are merged to become the aliases of the Net. The member expansions and connections are defined exactly as if there were 2 aliases on a single wire bus. In this case, there would be one NetBus A[0..1] with alias B[2..3} and 2 members A0 and A1.

The more normal method of connecting buses is not by direct physical connection as in the previous example, but by BusEntry. Connecting the 2 ends of a BusEntry to 2 different buses is NOT the same as a physical connection. Each WireBus is expanded independently of the and its members will connect to the scalar by name only.

Note: *This means that connecting two bus segments by BusEntry is not equivalent to connecting those segments through a 45 degree bus segment*.

- Case 8: A BusEntry connects two WireBuses with the same base name.

  WireBus with Alias A[0..7] expands to create Nets A0, A2, ... A7.

  WireBus with Alias A[4..7] expands to create Nets A4, A5, A6, A7.

  Since there is no physical connection, the members are connected by name only. The result is 2 NetBuses and 8 NetScalars:

  NetBus A[0..7] with members A0, A1, A2, A3, A4, A5, A6, A7
  NetBus A[4..7] with members A4, A5, A6, A7
  NetScalars A0, A1, A2, A3, A4, A5, A6, A7

- Case 9: A BusEntry connects two WireBuses with the different base names.

  WireBus with Alias A[0..7] expands to create Nets A0, A1, A2, ... A7.

  WireBus with Alias B[0..3] expands to create Nets B0, B1, B2, B3.

  Since there is no physical connection, the members are connected by name only. The result is that none of the member scalars are connected:

  NetBus A[0..7] with members A0, A1, A2, A3, A4, A5, A6, A7
  NetBus B[0..3] with members B0, B1, B2, B3
  NetScalars A0, A1, A2, A3, A4, A5, A6, A7, B0, B1, B2, B3

Aliases are also added as a result of connecting by name.

- Case10: A bus has aliases A[0..1] and B[2..3]. A second unconnected bus has aliases B[2..3] and C[0..7].

  As a result of having the same name B[2..3] the two buses are connected and all aliases are merged following the same rules as for aliases on a single wire. In this example the widest bus alias becomes the name of the NetBus and the remaining scalars are bitwise connected. The result is a NetBus C[0..7] with 2 aliases A[0..1]and B[2..3] and 8 members: A0, A1, C2, C3, ...C7. A0 has aliases B2 and C0. A1 has aliases B3 and C1.

Wire scalars do not connect to buses as a result of physical connection. Physically connecting a scalar to a bus will NOT change the connectivity in any way. Connecting one end of a BusEntry to a scalar wire and the other to a bus wire will NOT change connectivity. Scalars connect to members of the bus by name only.

When a wire scalar is placed a scalar net is created to represent that wire in the netlist. When one or more aliases are added to the wire, corresponding net aliases are created for that scalar net. The net scalars that correspond to the bus members are created following the rules above. Once created, the scalar net bus members connect to other scalar nets by name.

A bus has aliases A[0..1] and B[2..3]. A wire scalar is placed to connect physically to the bus. A scalar net with an automatically generated name (N0001 for example) is created to represent the wire. There is as yet no connectivity relationship between the scalar and the bus or its members. An alias C0 is added to the wire scalar. The scalar net now has name C0 but remains unconnected to any of the bus members regardless of the physical connection. A second alias, B2 is added to the wire. The Net scalar representing the wire is now connected to the bus member with alias B2. The resulting name of the wire's net is, therefore, A0.

See dbowire.h and dbobus.h

Alias

```
ENTITY Alias
SUBTYPE OF (BaseObject);
          Name:          StringT;
          Owner:         Wire;
          Location: Location;
          Rotation: RotationT;
          Font:          FontT;
          Color:         ColorT;
END_ENTITY;
```

The Alias defines an alias name for a Net and a graphical display of that name. The name of the Alias acts as an alias name for the Net associated with the Wire.

See dboalias.h

BusEntry

```
ENTITY BusEntry
SUBTYPE OF(BaseObject);
          Owner:    Page;
          EntryPoint:LocationT;
          EndPoint:LocationT;
          Color:    ColorT;
DERIVE
          EntryWire:Wire;
          EndWire:Wire;
END_ENTITY;
```

A BusEntry graphically represents the ripping a subrange or scalar from a WireBus. One end of the BusEntry connects with a WireBus and the other with another WireBus (a subrange of the 1st) or a WireScalar (a member of the bus). There is no constraint, however, that a BusEntry connect in such a manner.

In fact, a BusEntry connecting to a scalar at one end and a bus at the other causes no electrical connectivity. The scalar must have an alias that corresponds to one of the bus members for the connection to occur; such a connection would occur regardless of the BusEntry.

A BusEntry that connects to WireBuses at both ends connects the two buses as if they physically connected. See the description of Net below.

See DboEntry.h

NetSymbolInstance

```
ENTITY NetSymbolInstance

SUBTYPE OF SymbolInstance

ABSTRACT SUPERTYPE OF (Global, OffPageConnector, Port, ERC, BookMark, Titleblock)

          Symbol:Symbol;

          SourceSymbolName:StringT;

          SourceLibName:StringT;

DERIVE

          IsBus:    BooleanT;

          HotSpot:Location;

          Wire:     OPTIONAL Wire;

          Net:      OPTIONAL Net;

END_ENTITY;
```

A NetSymbolInstance may be placed on the Page and connected to a Wire in a manner similar to a PartInst. It is an instance of a NetSymbol just as a PartInst is an instance of a LibPart. The HotSpot is computed from the HotSpot of the Symbol, taking into account the location, rotation, and mirror of the instance. As a GraphicInstance, the NetSymbolInstance has a set of UserProps and a set of DisplayProps. As a SymbolInstance it has a list of pins. However, that list is limited to a single pin. Hence, the instance surfaces a number of derived attributes directly that are actually attributes of its single pin such as IsBus, HotSpot, Wire, and Net.

A NetSymbolInstance is used to add an attribute to another object on the page and to show that attribute visually. Hence, a Port connected to a Wire indicates that the signal on that Wire exits the Schematic and connects Nets higher in the design hierarchy. An ERC indicates that the object violated a design rule.

There are no different object types to differentiate bus and scalar instances. If the name has the format of a bus (<basename>[m..n]), then the instance is considered a bus; else it is a scalar. Physical connections of a bus instance to any other scalar entity cause no electrical connections. Similarly, physical connections of a scalar instance to a bus entity cause no electrical connection.

See dbosmbli.h

Global

```
ENTITY Global

SUBTYPE OF (NetSymbolInstance)

WHERE

          -- The defining Symbol is a GlobalSymbol

END_ENTITY;
```

A Global instance represents a global signal in the design. That means the Net will connect to any global Net anywhere else in the Design by name. There need be no Port-Pin connection through the hierarchy.

Creating a Global automatically creates a Net in the Page. The Global connects by name to any other Global, or global PortInst with that name. It also connects to any Wire with a matching Alias or any Port or OffPageConnector with the same name. If a Global is physically connected to a Wire, the Wire becomes connected to the global Net as well. The Name of the Global becomes the name of the Net.

If the Net also connects to a Port, the Global is “isolated” to the schematic; it connects outside the schematic only through the Port.

If the Net connects to an OffPageConnector, the Global is “isolated” to the page; it connects outside the page only through the OffPageConnector.

See dbosmbli.h

OffPageConnector

```
ENTITY OffPageConnector

SUBTYPE OF (NetSymbolInstance)

WHERE

          -- The defining Symbol is an OffPageConnectorSymbol
```

```
END_ENTITY;
```

An OffPageConnector permits nets to connect by name across Page boundaries of the same schematic. If there are two Nets on different Pages of a Schematic that have the same name, they will not be part of the same SchematicNet unless there is an OffPageConnector (or Port) on both pages

Creating an OffPageConnector automatically creates a Net in the Page. The OffPageConnector connects by name to any other OffPageConnector, Port, or Global with the same name. It also connects to any Wire with a matching Alias. If an OffPageConnector is physically connected to a Wire, the Wire becomes connected to the connector’s Net. The Name of the OffPageConnector becomes the name of the Net. (Note a Port on the same Net will override the connector’s name.)

.See dbosmbli.h

Port

```
ENTITY Port

SUBTYPE OF (NetSymbolInstance)

          Type:     PinTypeT;

WHERE

          -- The defining Symbol is a PortSymbol

END_ENTITY;
```

A Port permits nets at one level to connect to nets higher in the design hierarchy. The Type indicates the type of signal flow. The set of Ports in a Schematic defines its “interface”.

Creating an Port automatically creates a Net in the Page. The Port connects by name to any other Port, OffPageConnector, or Global in the Page with the same name. It also connects to any Wire with a matching Alias. If a Port is physically connected to a Wire, the Wire becomes connected to the Port’s Net. The name of the Port becomes the name of the Net.

See dbosmbli.h

GraphicBoxInst

```
ENTITY GraphicBoxInst
```

```
SUBTYPE OF (GraphicInstance)

WHERE

          - constrained to consist of a single Box Vector

END_ENTITY;
```

A GraphicBoxInst is a GraphicInstance whose defining GraphicObject consists of a single Box.

See dbogrphi.h

GraphicLineInst

```
ENTITY GraphicLineInst

SUBTYPE OF (GraphicInstance)

WHERE

          - constrained to consist of a single Line Vector

END_ENTITY;
```

A GraphicLineInst is a GraphicInstance whose defining GraphicObject consists of a single Line.

See dbogrphi.h

GraphicArcInst

```
ENTITY GraphicArcInst

SUBTYPE OF (GraphicInstance)

WHERE

          - constrained to consist of a single Arc Vector

END_ENTITY;
```

A GraphicArcInst is a GraphicInstance whose defining GraphicObject consists of a single Arc.

See dbogrphi.h

GraphicEllipseInst

```
ENTITY GraphicEllipseInst
SUBTYPE OF (GraphicInstance)
WHERE
          - constrained to consist of a single Ellipse Vector
END_ENTITY;
```

A GraphicEllipseInst is a GraphicInstance whose defining GraphicObject consists of a single Ellipse.

See dbogrphi.h

GraphicPolygonInst

```
ENTITY GraphicPolygonInst
SUBTYPE OF (GraphicInstance)
```

WHERE

- constrained to consist of a single Polygon Vector

```
END_ENTITY;
```

A GraphicPolygonInst is a GraphicInstance whose defining GraphicObject consists of a single Polygon.

See dbogrphi.h

GraphicPolylineInst

```
ENTITY GraphicPolylineInst
SUBTYPE OF (GraphicInstance)
```

WHERE

- constrained to consist of a single Polyline Vector

```
END_ENTITY;
```

A GraphicPolylineInst is a GraphicInstance whose defining GraphicObject consists of a single Polyline.

See dbogrphi.h

GraphicCommentTextInst

```
ENTITY GraphicCommentTextInst
SUBTYPE OF (GraphicInstance)
```

WHERE

- constrained to consist of a single Text Vector

```
END_ENTITY;
```

A GraphicCommentTextInst is a GraphicInstance whose defining GraphicObject consists of a single Text Vector.

See dbogrphi.h

GraphicBitMapInst

```
ENTITY GraphicBitMapInst
SUBTYPE OF (GraphicInstance)
```

WHERE

- constrained to consist of a single BitMap Vector

```
END_ENTITY;
```

A GraphicBitMapInst is a GraphicInstance whose defining GraphicObject consists of a single BitMap Vector.

See dbogrphi.h

GraphicSymbolVectorInst

```
ENTITY GraphicSymbolVectorInst

SUBTYPE OF (GraphicInstance)
```

WHERE

- constrained to consist of a single SymbolVector

```
END_ENTITY;
```

A GraphicSymbolVectorInst is a GraphicInstance whose defining GraphicObject consists of a single SymbolVector. (This is probably redundant with the GraphicObject, but seemed convenient).

See dbogrphi.h

ParameterInstance

```
ENTITY ParameterInstance

SUBTYPE OF (SymbolInstance)
```

WHERE

The defining Symbol is a ParameterSymbol

The set of UserProps is the set of UserProps for the owning schematic

```
END_ENTITY;
```

A ParameterInstance represents the properties of the schematic. It provides a graphical method of displaying and editing those properties. A ParameterInstance will not carry any UserProps of its own. Adding a UserProp to a ParameterInstanceactually adds that property to the schematic that owns it. Adding a DisplayProp to a Parameter displays a schematic property.

See Psmbli.h

OptimizerInstance

```
ENTITY OptimizerInstance

SUBTYPE OF (SymbolInstance)

DERIVE

          OptimizerParameters: SET [0:?] OF OptimizerParameter

          UseOptimizedValues: BooleanT
```

WHERE

-- The defining Symbol is an OptimizerSymbol

```
END_ENTITY;
```

An OptimizerInstance provides a graphical means of displaying OptimizerParameters associated with a schematic. It is restricted to have no UserProps but otherwise it inherits all the basic attributes of the SymbolInstnce. The OptimizerInstance also provides access to the set of OptimizerParameters stored with the Schematic. The instance exists to manage the display of those parameters. In addition, a boolean value, UseOptimizedValues, indicates whether the PSpice simulator should use the optimized values or the current values for the set of OptimizerParameters.

See Psmbli.h

OptimizerParameter

```
ENTITY OptimizerParameter

SUBTYPE OF (BaseObject);

          Name:     StringT;
          Owner:    Schematic;
          InitialValue:StringT;
          CurrentValue:StringT;
          LowerLimit:StringT;
          UpperLimit:StringT;
          Tolerance:StringT;
```

WHERE

Name is unique for all OptimizerParameters within the owning schematic.

```
END_ENTITY;
```

An OptimizerParameter associates parameters required by the optimizer with an owning schematic. The Name of the OptimizerParameter must be unique within the schematic.

See Psmbli.h

Stimulus

```
ENTITY Stimulus
SUBTYPE OF (SymbolInstance)
          Stimulus:      StringT;
```

WHERE

-- The defining Symbol is a StimulusSymbol

```
END_ENTITY;
```

A Stimulus instance represents a stimulus source into the design for use during PSpice simulation. The stimulus is associated with a signal in the design by connecting the pins of the Stimulus to wires or pins in the schematic page. The Stimulus attribute is the name of an entry within a stimulus library (.stl file). The path to the file is not specified, but is determined by a searchpath held in the owing design or library.

See Psmbli.h

PSpiceSource

```
ENTITY PSpiceSource

SUBTYPE OF (SymbolInstance);

WHERE

          -- The defining Symbol is a PSpiceSourceSymbol

END_ENTITY;
```

The PSpiceSource instance is derived from SymbolInstance It represents a signal source in the design used during PSpice simulation. The source is associated with a signal in the design by connecting the pins of the PSpiceSource instance to wires or pins in the schematic page.

See Psmbli.h

PSpiceABMInstance

```
ENTITY PSpiceABMInstance

SUBTYPE OF (SymbolInstance);

WHERE

          -- The defining Symbol is a PSpiceABMSymbol

END_ENTITY;
```

The PSpiceABMInstance is derived from SymbolInstance. This provides it all the graphical characteristics of a GraphicInst and a list of pins. It represents an instance in the design whose PSpice simulation characteristics are defined by an Analog Behavioral Model. It connects to signals in the design be connecting its pins to wires or other pins in the schematic page.

See Psmbli.h

PSpiceSimulationDirective

```
ENTITY PSpiceSimulationDirective

SUBTYPE OF (SymbolInstance);

WHERE
```

```
          -- The defining symbol is a PSpiceSimulationDirectiveSymbol
END_ENTITY;
```

The PspiceSimulationDirective instance is derived from SymbolInstance. This provides it all the graphical characteristics of a GraphicInst and a list of pins. It provides instructions to the PSpice simulator. Connecting the pins of the PSpiceSimulationDirective to wires or pins in the design informs the simulator about any signals on which the instruction is to be executed.

See Psmbli.h

Net

```
ENTITY Net
SUBTYPE OF (BaseObject);
ABSTRACT SUPERTYPE OF (NetScalar, NetBus);
DERIVE
          Name:     OPTIONAL StringT;
          NetName:  StringT;
          Owner:    Schematic;
          Aliases:  SET [0:?] OF StringT;
          Wires:    SET [0:?] OF Wire;
          PortInsts:SET [0:?] OF PortInst;
          Ports:    SET [0:?] OF Port;
          Globals:  SET [0:?] OF Global;
          OffPageConnectors: SET [0:?] OF OffPageConnector;
WHERE
          -- Name is one of the strings in the set of Aliases
END_ENTITY;
```

A Net represents a page-wide electrical signal. The Net may be either a single signal, a NetScalar, or a group of signals, a NetBus. It is not “complete” in the sense of including all the pages in the Schematic (see SchematicNet) or in the Design (see FlatNet). It cannot be used for netlisting or DRC. The Net is computed as the first step in computing the SchematicNet. It is used during interactive Page editing to avoid the memory requirements and performance cost of maintaining SchematicNets interactively. Such Nets may not have the final net name if there are errors in OffPageConnector placement or may not have the complete list of properties found on the SchematicNet.

A Net consists of a set of Wires, PortInsts, Ports, Globals, and OffPageConnectors which are connected by physical connection, Name, Alias, BusEntry, or Global Name within a Page

The set of Aliases for a Net is computed from the objects that it contains:

- Names of all Aliases on all Wires in its set of Wires
- Names of all Ports in its set of Ports
- Names of all Globals in its set of Globals

- Names of all OffPageConnectors in its set of OffPageConnectors
- If there are no other objects in the Net, Name of a PartInst+PortInst in its set of PortInsts

A Net optionally has a user specified Name. The Name must be among the set of Aliases (minus the PortInst Name). This gives the user a means of overriding the normally computed NetName.

Each Net has a computed NetName which is determined by the Names and Aliases of its constituent parts. If there is a user specified Name, use that as the NetName. If there is more than one name with the same priority, use the following rules: If the Net is a bus and there is more than one Alias, use the Alias busname with the largest width (range). If there is more than one busname with the same width, use the alphabetically first. If the Net is a scalar, use the Alias name which is alphabetically first. The following rules list the priority of aliases when

1. If there is a Port on the Net use the name of the Port.
2. If there is an OffPageConnector on the Net use the name of the OffPageConnector.
3. If there is a Global or global PortInst on the Net, use the name of the Global.
4. If there is an Alias on the set of Wires, use the name of the Alias.
5. If there is no “source” of a name but the Net contains a Wire, generate a name of the form Nxxxxx, where xxxxx is the unique Wire id with enough leading zeros to fill 5 digits.
6. If there is no “source” of a name but the Net contains a PortInst, generate a name of the form Nxxxxxyyy, where xxxxx is the unique PartInst id with enough leading zeros to fill 5 digits and yyy is the position of the PortInst in the pin list for that instance.

UserProps and DisplayProps are not permitted on a Net. The EffectiveProps are simple (Name,Value) pairs determined by merging the UserProps on all the Wires, Globals, OffPageConnectors, and Ports that make up the Net. If there are conflicting UserProps such that two UserProps have the same Name and different Values, the Value is the alphabetically first among the set of conflicting Values.

See WireScalar and WireBus for further description of “connection” Nets.

See dbonet.h.

NetScalar

```
ENTITY NetScalar

SUBTYPE OF (Net)

END_ENTITY;
```

A NetScalar represents a single-bit signal on the schematic.

See dbonet.h

NetBus

```
ENTITY NetBus

SUBTYPE OF (Net)

DERIVE

          BaseName:StringT;

          LSB       Int;

          MSB       Int;

          Nets:SET [0:?] OF Net;

END_ENTITY;
```

A NetBus is a grouping of Nets in which the members carry the same base name as the containing bus and are identified by index. The range of these indices is specified by the LSB (least significant bit) and MSB (most significant bit) values.

See section WireBus above for further information about the computation of NetBuses.

See dbonet.h and dbobus.h

Schematic Objects

Schematic

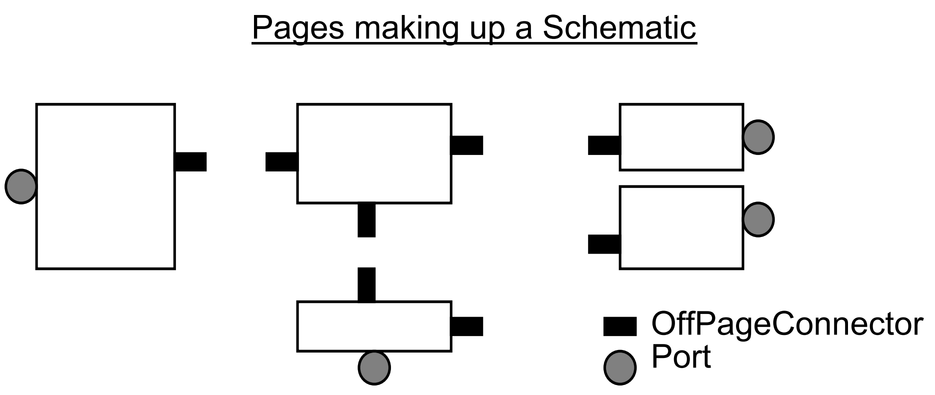
```
ENTITY Schematic
SUBTYPE OF (View);
          Pages:          LIST [0:?] OF Page;
          OptimizerParameters:SET [0:?] OF OptimizerParameter;
          UseOptimizedValues: BooleanT
          UserStorageIStorage;
DERIVE
          Insts:    SET [0:?] OF PartInst;
          Nets:     SET [0:?] OF SchematicNet;
          Ports:    SET [0:?] OF SchematicPort;
          Globals:SET [0:?] OF SchematicGlobal;
          OffPageConnectors: SET [0:?] OF SchematicOffpageConnectors
END_ENTITY;
```

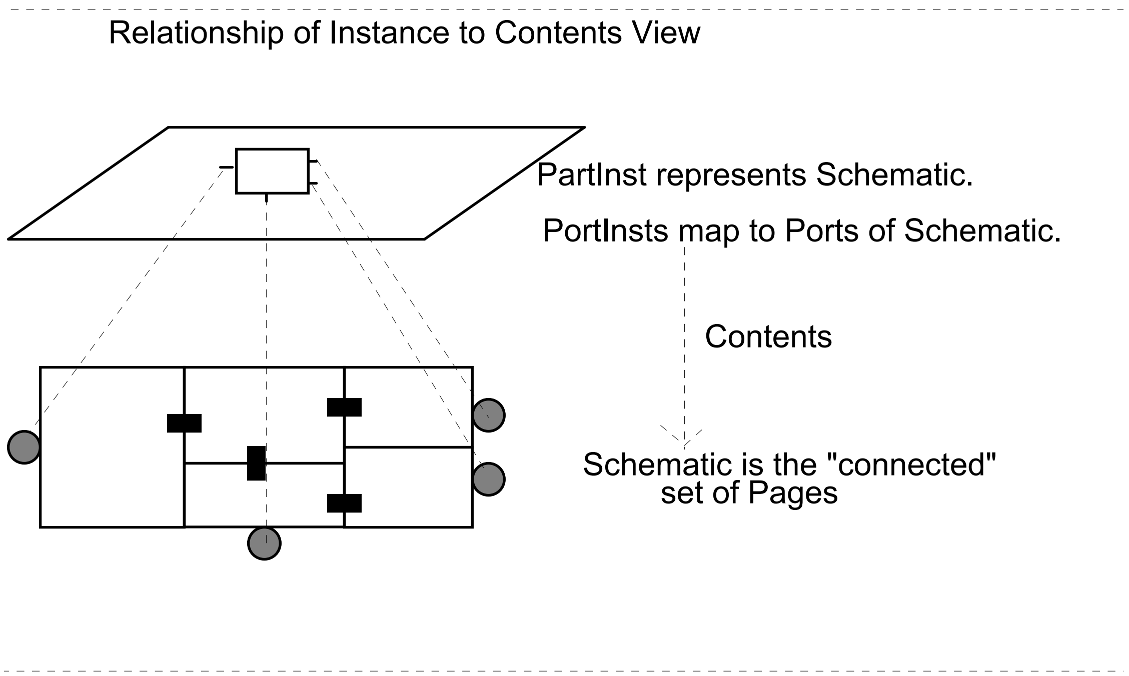
A single Page may not provide the complete electrical content of an electrical object. For reasons of print size or information complexity, the designer may choose to use multiple Pages to describe a single object. The Schematic groups sets of Pages together to provide a complete graphical description of an electrical object.

A Schematic also provides access to the netlist connectivity described by its set of Pages. It is the Schematic which maintains knowledge of the electrical SchematicNets represented by the Wires, Ports, ... A Schematic Net represents the electrical signal made up of all the Wires connected by physical connection or Name within a Page or by OffPageConnector across sheet boundaries. The Schematic also understands the relationships represented by buses. The Schematic Nets include the individual scalars that are computed from the range specified by a bus name. For example, a WireBus may have an Alias of B[0..7]. The Schematic not only contains SchematicNetBus B[0..7] but also NetScalars B0, B1...B7. It also understands that B1..B7 are all members of B[0..7].

The Schematic provides the netlist connectivity appropriate to a hierarchical netlister. It is the Schematic as a whole which defines the Contents under a PartInst, not a single Page. The PortInsts of the PartInst correspond to the SchematicPorts of the Schematic hierarchically below (computed from all the Ports on all Pages of the Schematic). There is no one Page of the Schematic which is a master or otherwise provides entry into the set; the Schematic object itself provides the containment.

Ports define signals that exit the schematic to the Design; they connect hierarchically. OffPageConnectors define signals that exit a page to connect to signals on other pages of the same Schematic; they connect page to page.




Relationship of Instance to Contents View
PartInst represents Schematic.
PortInsts map to Ports of Schematic.
Contents
Schematic is the "connected" set of Pages

A Schematic is always contained within a Library or Design. A Page is always contained within a Schematic. In many cases the Design will consist of a single Schematic which contains a single Page. The Design may, however, contain a Schematic at its root whose instances reference other Schematics as their Contents. Any of these Schematics may contain multiple Pages.

A Schematic is type of View. It defines the netlist connectivity described graphically by a set of Pages. Because the connectivity of the Schematic is derived from the Pages that make it up, there are no functions to create Insts, Nets, or Ports.

The Insts correspond one-to-one to the PartInsts on all the Pages that make up the schematic.

The SchematicNets do not correspond in such a simple one-to-one fashion. A Net of the Schematic represents an electrical signal at a single level of the design hierarchy. It consists of a set of Wires, Ports, Globals, OffPageConnectors, and PortInsts in its pages. SchematicNets are computed by connecting the page-wide nets across page boundaries based on Global, Port, and OffPageConnector names.

The SchematicPorts correspond to Ports in the Pages, but this relation may also not be a simple 1-1 relationship. A Port may represent a Bus. If so, single-bit PortScalars may exist in the Schematic which have no corresponding Port in any of the Pages. There may be multiple Ports on the various Pages of the Schematic which have the same Name. These are all merged into a single Port entity in the Schematic. SchematicGlobals and OffPageConnectors have similar relationships with their sources on the Page.

Much of the contents of the Schematic are computed from the graphical objects contained within its pages. This computation is not maintained continuously to avoid the necessity of reading in all the Pages. Before accessing any of these objects the application must Expand() the Schematic. The application can query the Schematic with IsExpanded() to determine if it has been previously expanded or to determine if a new expansion is required because of edits.

A Schematic is ultimately derived from DboBaseObject and, therefore, has a set of UserProps. These UserProps can be accessed directly through the Schematic or may be accessed through a ParameterInstance on any of the Schematic’s pages.

The Schematic also contains a set of OptimizerParamaters. These parameters are used by PSpice to manage its optimization process. An additional boolean attribute, UseOptimizedValues, indicates that PSpice should use the Optimized values saved in these parameters when simulating. These values can be accessed directly through the Schematic or may be accessed through the OptimizerParameterInstance on any of the Schematic’s pages.

A UserStorage area is provided for the use of applications to store additional data associated with the schematic but which is not directly managed by the DB. The data within the storage is not available unless that application is running; standard Capture does not “understand” it. It is essentially a very large UserProp attached to the schematic.

See dboschm.h

SchematicNet

```
ENTITY SchematicNet
SUBTYPE OF (BaseObject);
ABSTRACT SUPERTYPE OF (SchematicNetScalar, SchematicNetBus);
          ID:            int;
DERIVE
          Name:     StringT;
          Owner:    Schematic;
          Aliases:  SET [0:?] OF StringT;
          Wires:    SET [0:?] OF Wire;
          PortInsts:SET [0:?] OF PortInst;
          Ports:    SET [0:?] OF SchematicPort;
          Globals:  SET [0:?] OF SchematicGlobal;
          OffPageConnectors: SET [0:?] OF SchematicOffPageConnector;
END_ENTITY;
```

A Net represents the schematic-wide signal. It may be either a single signal, a SchematicNetScalar, or a group of signals, a SchematicNetBus. It is an entity computed by connecting the page-wide Nets within all the pages of the Schematic. Nets are combined into a single SchematicNet if:

1) Both Nets contain a Global or a global PortInst with the same name. An exception is if either of the Nets also contains an OffPageConnector. If so, that Net is isolated to its page and will not connect globally.

2) Both Nets contain a Port or OffPageConnector with the same name.

The Name of the Schematic Net is normally the same as that of the page-wide nets that define it. It must be unique within the Schematic. If the user has not placed OffPageConnectors appropriately, there may be cases where two page-wide Nets do not connect across page boundaries even though they have the same name. In such cases a unique number is appended to the page-wide Net name. The number is the smallest ID of all the Nets that make up the SchematicNet.

The set of PortInsts is computed from all those PortInsts connected to any of the page-wide Nets that make up the SchematicNet; these PortInsts are all "connected" by this net and may not be part of any other Net.

As a computed object, the SchematicNet does not carry UserProps. Since it is not displayed, it has no DisplayProps. The UserProps are simple (Name,Value) pairs determined by merging the UserProps on all the Wires and Ports that make up the Nets that make up the SchematicNet. If there are conflicting UserProps such that two UserProps have the same Name and different Values, the Value returned is the lexically smallest Value.

See dboschmn.h

SchematicNetScalar

```
ENTITY SchematicNetScalar
SUBTYPE OF (SchematicNet)
DERIVE
END_ENTITY;
```

A SchematicNetScalar represents a single-bit signal on the schematic.

See dboschmn.h

SchematicNetBus

```
ENTITY SchematicNetBus
SUBTYPE OF (SchematicNet)
DERIVE
          BaseName:StringT;
          LSB       Int;
          MSB       Int;
          Members:LIST [0:?] OF SchematicNetScalar;
END_ENTITY;
```

A NetBus is a grouping of Nets in which the members carry the same base name as the containing bus and are identified by index. The range of these indices is specified by the LSB and MSB values.

See dboschm.h and dbobus.h

SchematicSymbolInst

```
ENTITY SchematicSymbolInst
SUBTYPE OF (BaseObject)
ABSTRACT SUPERTYPE OF (SchematicPort, SchematicGlobal, SchematicOffPageConnector);
DERIVE
          Name:     StringT;
          Owner:    Schematic;
          Net:      SchematicNet;
          Entries:  SET [0:?] OF BaseObject;
          IsBus:    BooleanT;
END_ENTITY;
```

The SchematicSymbolInst is an abstract object. It provides the base class for the SchematicPort, SchematicGlobal and the SchematicOffPageConnector. It is a single object in the Schematic which is computed from possibly many sources in the Schematic’s Pages.

See dboschmi.h

SchematicPort

```
ENTITY SchematicPort
SUBTYPE OF (SchematicSymbolInst)
DERIVE
          PinType:PinTypeT;
END_ENTITY;
```

The SchematicPorts define the “interface” of a Schematic. These are the signals that connect by name to PortInsts on the instance at the next higher level of the hierarchy.

See dboschmi.h

SchematicGlobal

```
ENTITY SchematicGlobal

SUBTYPE OF (SchematicSymbolInst)

DERIVE

END_ENTITY;
```

The SchematicGlobals define the global signals of a Schematic. These are the signals that connect by name to other global signals throughout the design. Such signals are normally Power type but are not restricted to that type.

See dboschmi.h

SchematicOffPageConnector

```
ENTITY SchematicOffPageConnector

SUBTYPE OF (SchematicSymbolInst)

DERIVE

END_ENTITY;
```

The SchematicOffPageConnectors define the cross-page connector of the Schematic. These are of little use other than in the computation of the SchematicNets themselves.

See dboschmi.h

Design Objects

Design

```
ENTITY Design;
SUBTYPE OF (Library);
         Root:          OPTIIONAL View;
DERIVE
         RootOccurrence:OPTIONAL InstOccurrence;
         Occurrences:SET [0:?] OF Occurrence;
         FlatNets: OPTIONAL SET [0:?] OF FlatNet;
END_ENTITY;
```

The Design object models an electrical design. It is a container, but it differs from a Library rather as a house differs from the pile of lumber, pipes, nails, etc. that are heaped up at a lumber yard. The Design constructs the hierarchy or "structure" of the final electrical product from the raw materials of the Library.

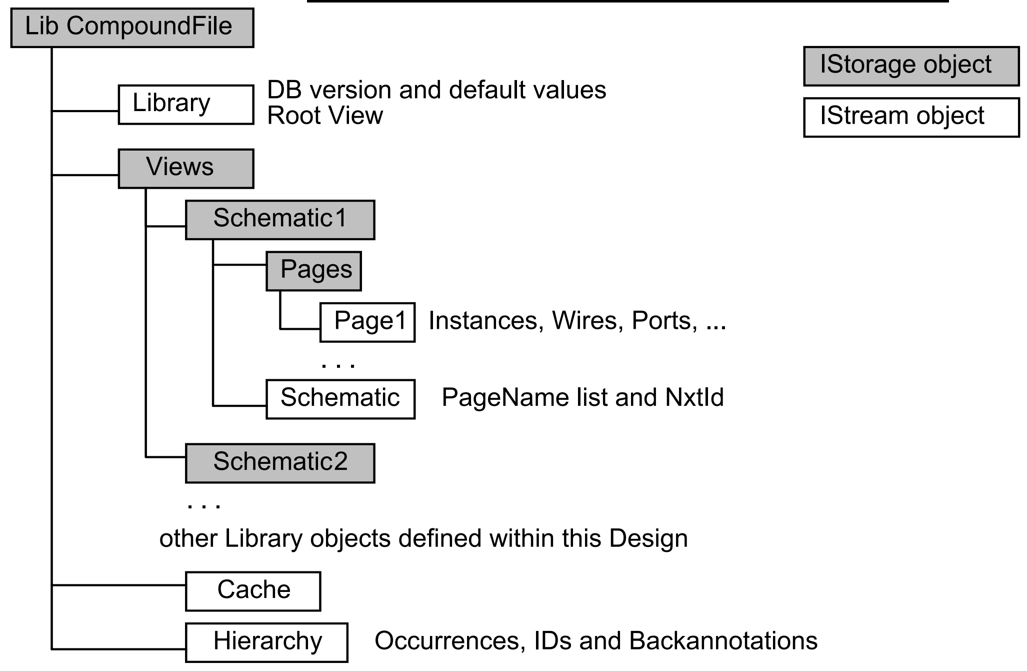
The Design contains all the Parts, Symbols, and Schematics which have been created specifically for this design, and it identifies one specific Schematic as the root of the design hierarchy. The Design also contains annotations and back annotations for this design. The packaging information for PCB layout, for example, would be held as annotated properties on the Occurrences within the Design. This information cannot be added directly on the Schematic because there are cases where a Schematic is instanced more than once in a Design. Consider the case of the Fulladd which contains two instances of Halfadd. It is not possible to change the Reference Designator or Pin Numbers on Halfadd directly since there will be two different sets of values.

A Design "contains" its Schematics, Pages, Parts and Symbols in a relationship which is the same as the Library and its contents. Its contents exist within the Design, move with the Design, and are deleted with the Design, but each is independently editable and savable.

The contents of a Design are highly interrelated. In particular, the hierarchy tree is completely computed from information in the various Schematics. In order to manage the integrity of these interrelated objects it is highly desirable that they all be maintained within a single physical file. Such a physical representation also permits a more compact storage mechanism than separate files could provide. It will be possible to load only those portions of the Design that the user wishes to view or edit; and it will be possible to save a Page or Part without updating all the other Design objects which are currently loaded.

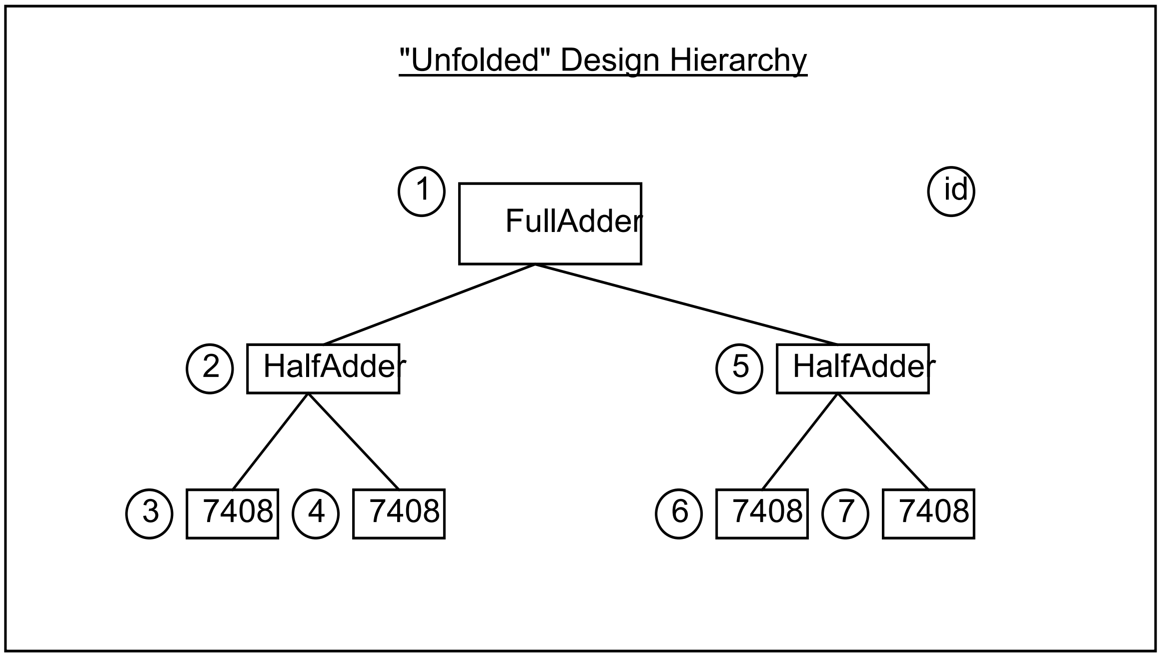
The layout of the Design file is similar to that of the Library, but it contains additional structures to maintain the hierarchy tree and its associated back annotation information.

Design's Representation as a Compound File

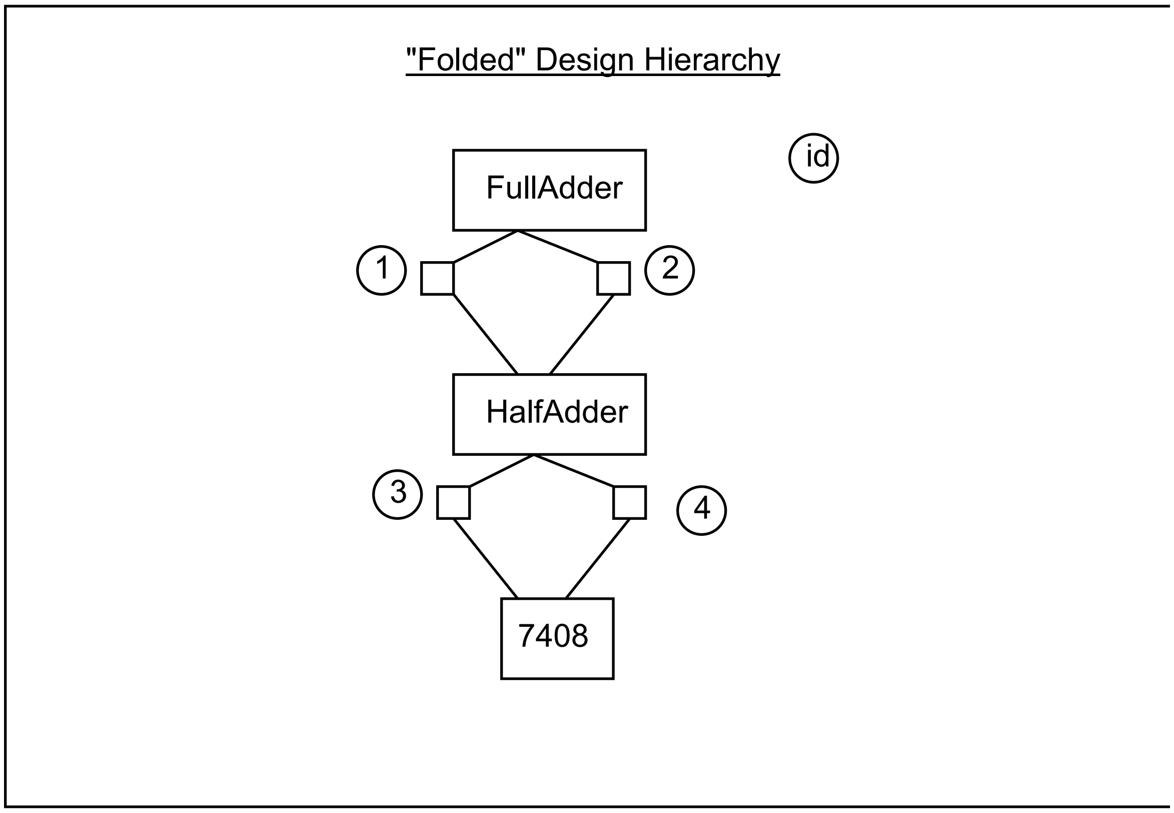


The Occurrences of a Design model the hierarchy of a design. Each non-leaf node in the hierarchy represents a Schematic which may have multiple Pages. It also supports a single one-Page schematic design or a flat multi-Page schematic design. The root of the Design is the *Root* Schematic.

The Occurrences of the Design permit a fully "unfolded" hierarchy. Such a hierarchy is represented by a tree in which each node has a unique identity even though it may represent a View which is reused. Consider the FullAdder in which the root Schematic contains two HalfAdders, and the HalfAdder contains two 7408's. A fully unfolded representation of that design is:



A "folded" representation of the Design would be a graph structure.



Most hierarchical netlists have a form which corresponds to the folded model. However, a fully unfolded model is required to compute a flattened design and an unfolded model is required to maintain backannotated layout information.

The FlatNets represent the electrical signals of the flattened design. A single FlatNet consists of a set of NetOccurrences in the design. The "primitive" Occurrences correspond to the instances of the flattened design; they have no children under them. Traversal of the FlatNets and primitives of the Design provide the information for a flattened netlist from the same structure that is traversed by a hierarchical netlister.

In order to access the Occurrence tree, the application should call GetRootOccurrence(). This returns the root InstOccurrence of the hierarchy. From there the application can traverse the hierarchy in a recursive tree walk. If the hierarchy has not yet been constructed, GetRootOccurrence will construct it.

The Occurrences of the Design are computed entities. They cannot be created by the application directly. They are not continuously maintained by the database, but are computed upon request. Any edit to the connectivity of the design invalidates the Occurrences. The application can determine if the occurrence tree is valid by calling GetOccurrencesAreValid (); GetRootOccurrence() will construct the tree if it has never yet been computed, but cannot be used to recompute it. If the application edits the design after computing the tree, it must call UpdateOccurrences() or NewRootOccurrence() to recompute the new tree. RemoveOccurrences() removes the tree from memory and DeleteOccurrences() will also remove the disk image.

The FlatNets of the Design are also computed entities. They are also computed only upon request by the application. If they have not yet been computed, starting a DboDesignFlatNetsIter will compute them. Once computed they remain valid as long as the Occurrences do. Once the design is modified, they must be recomputed through a call to UpdateFlatNets(). RemoveFlatNets removes the nets from memory. Unlike Occurrences, FlatNets are not persistent.

See dbodsgn.h

Backannotation

```
ENTITY Backannotation
SUBTYPE OF (UserProp);
WHERE
          - owner is Occurrence
END_ENTITY
```

A Backannotation is simply a UserProp on an Occurrence.

See dbobacka.h

Occurrence

```
ENTITY Occurrence
ABSTRACT SUPERTYPE OF (InstOccurrence,
                       NetOccurrence,
                       PortOccurrence,
                       TitleBlockOccurrence);
          Owner:         Design;
          Pathname: StringT;
          Id:            int;
          Parent:        Occurrence;
          Backannotations:SET [0:?] OF Backannotation;
END_ENTITY;
```

An Occurrence is an abstract entity that represents a node in the hierarchy tree. It may be either an Instance, Net, or Port within a Schematic of that Design. Each Occurrence has an Id that is unique within the entire Design. Its Pathname is the concatenation of instance names that defines a unique path through the tree; the Pathname may also be used as a unique identifier.

The Backannotations added to an Occurrence carry occurrence-specific information on the design. Designs in which a Schematic is reused must use Backannotations rather than UserProps on the Schematic object, since different values are required for the multiple different occurrences of those objects.

See dboocc.h

InstOccurrence

```
ENTITY InstOccurrence
SUBTYPE OF (Occurrence)
          Inst:     OPTIONAL PartInst;
          Reference:StringT;
          Designator:StringT;
          Children:SET [0:?] OF Occurrence;
DERIVE
          Schematic:OPTIONAL Schematic;
          Contents:OPTIONAL View:
          IsPrimitive:BooleanT;
          ReferenceDesignator: StringT;
END_ENTITY;
```

An InstOccurrence represents the occurrence of a PartInst within the fully unfolded hierarchy of the Design. If the InstOccurrence is non-primitive, the InstOccurrence might also be considered a “SchematicOccurrence” since there will also be a Schematic associated with any hierarchical occurrence (the Contents attribute).

Its Reference and Designator are occurrence specific values normally set by the Annotate tool but which may be manually set. In a Design where a Schematic is reused, the Reference and Designator must be set on the InstOccurrence rather than the PartInst since they must be unique in the expanded Design.

The Schematic of an InstOccurrence is the Schematic that owns the source Inst. The Contents is the Contents of the source Inst. If the Contents is NULL, the InstOccurrence is primitive. If the source Inst has been marked IsPrimitive, the InstOccurrence is primitive regardless of whether a ContentsView has been specified.

The Children of an InstOccurrence are those InstOccurrences, NetOccurrences, and TitleBlockOccurrences that correspond to the PartInsts, SchematicNets, and TitleBlocks in the Contents View. A primitive InstOccurrence has an empty set of Inst, Net, and TitleBlockOccurrence Children.

The PortOccurrence children correspond to the PortInsts of the source Inst. For a non-primitive InstOccurrence, there are normally also Ports in the Contents View that match by name to these PortInsts. If there is a mismatch of PortInsts to Ports, there will be a unique PortOccurrence for each unique name when the Ports and PortInsts are combined. For example, if a hierarchical PartInst has a PortInst named “IN” and there is no such Port in its Contents View, there will be a PortOccurrence with the PortInst as its source. Similarly, if there is a Port in the Contents View named “OUT” but no PortInst on the Inst, there will be a PortOccurrence with the Port as its source.

Given an InstOccurrence and an object in its Contents, it is possible to get the child occurrence that is associated with that object. This permits the application to move into the Schematic or Page to get associations that are not directly available through the Occurrence and then link the new object back up into the tree. See the functions GetInstOccurrence(), GetPortOccurrence(), and GetNetOccurrence().

The root Occurrence of the Design is an InstOccurrence whose Inst is NULL and Parent is the Design. Its Contents is the Root View of the Design.

The EffectiveProps of an InstOccurrence are computed by merging the DBProps and UserProps of the Occurrence with the UserProps of the source Inst and the UserProps of the underlying LibPart. Properties on the Occurrence take precedence over the instance which takes precedence over the LibPart.

See dboinsto.h

NetOccurrence

```
ENTITY NetOccurrence
SUBTYPE OF (Occurrence)
          Net:      Net;
          FlatNet:  FlatNet;
END_ENTITY;
```

A NetOccurrence represents the occurrence of a schematic Net within the fully unfolded hierarchy of the Design. Every NetOccurrence will be part of exactly one FlatNet.

See dboneto.h

PortOccurrence

```
ENTITY PortOccurrence

SUBTYPE OF (Occurrence)

          PortInst:OPTIONAL PortInst:

          Port:     OPTIONAL Port;

          FlatNet:  FlatNet;

END_ENTITY;
```

A PortOccurrence represents both the Port and the PortInst above in the design hierarchy. For those primitive occurrences (those which have no Children), the Port will not exist. For those PortOccurrences corresponding to Ports in the Root of the Design, there will be no PortInst.

See dboporto.h

FlatNet

```
ENTITY FlatNet

SUBTYPE  OF (BaseObject);

          Name:          StringT;

          NetOccurrences:SET [0:?] OF NetOccurrence;

          TopNet:   NetOccurrence;

END_ENTITY;
```

A FlatNet represents a single electrical signal in the fully bound electrical design. It collects together the complete set of NetOccurrences which are "connected" through Port/Pin mappings and through global net names according to the following rules:

1. If NetOccurrences N1 and N2 are both connected to the same PortOccurrence (one is connected to the Port and one is connected to the PortInst above), then N1 is connected to N2.
2. If NetOccurrences N1 and N2 have a global Name in common, then N1 is connected to N2.

The TopNet acts as a "representative" net for the set. It is the TopNet's Name which is typically used to identify the FlatNet when producing a flattened netlist. The TopNet is determined by the following rules:

1. Set the SearchSet = the set of NetOccurrences of the FlatNet.
2. If there are no globals in the SearchSet, go on to step 3. If there is exactly one NetOccurrence in the SearchSet which is global, select that NetOccurrence. Else reduce the SearchSet to the set of NetOccurrences which are global.
3. If there is one NetOccurrence in the SearchSet whose Path is highest in the hierarchy, select that NetOccurrence. Else reduce the SearchSet to the set of NetOccurrences which have the shortest Path (are highest in the design hierarchy).
4. Select the NetOccurrence which is first in an alphabetical sorting of string in Path, first to last.

As a computed entity, the FlatNet has no UserProps or DisplayProps.

The EffectiveProps of the FlatNet represent the merged UserProps of all the NetOccurrences that make it up. In case there are conflicts of value for UserProps with the same Name, the alphabetically first value is selected.

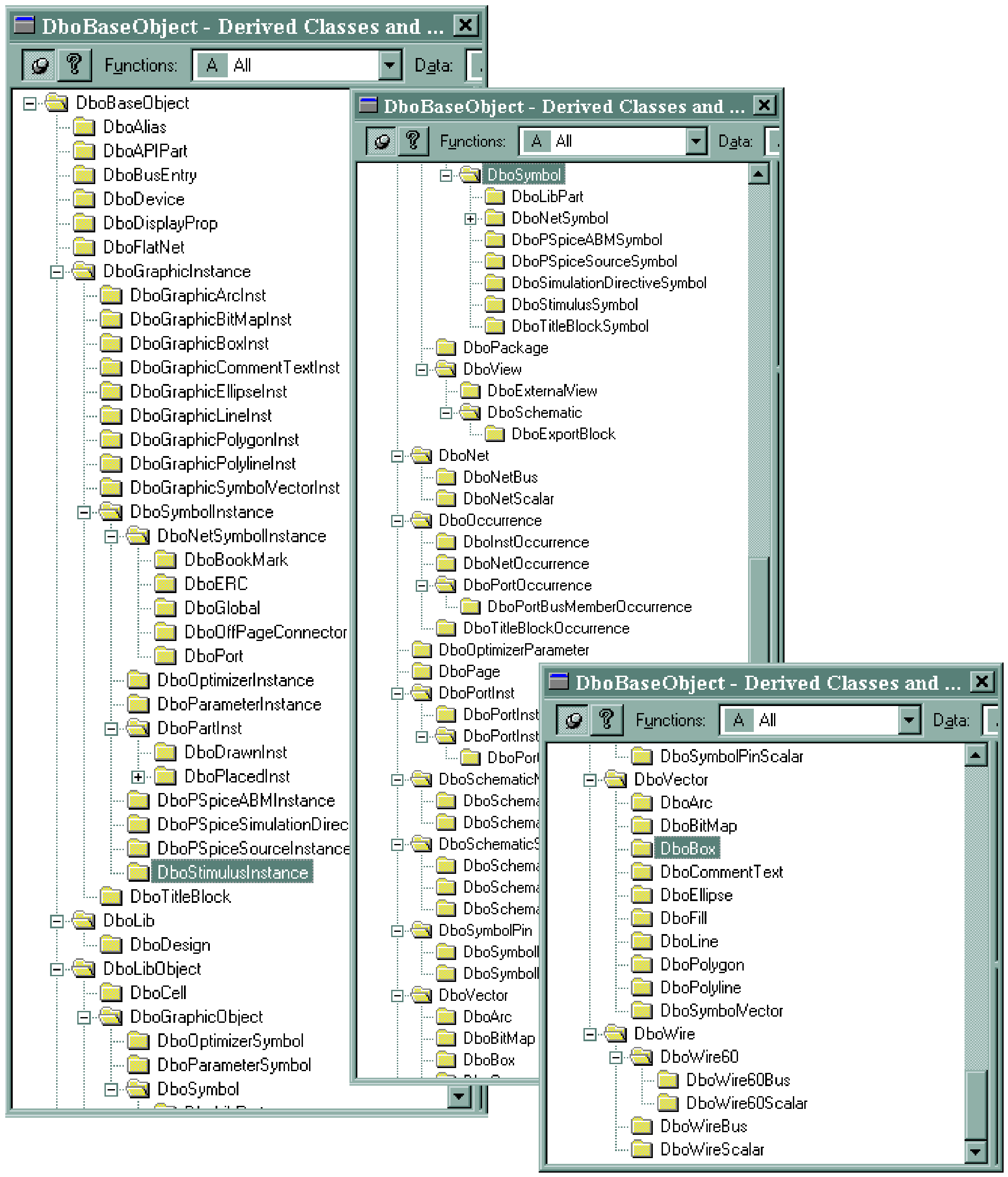
UserStorage

```
ENTITY UserLibStorage

          Owner     Library

          Name:     StringT;

DERIVE:

          Storage   IStorageT

END_ENTITY
```

The UserStorage is a storage area within a library, schematic or page into which a user may write data. This format of this data is unknown to Capture. Capture merely provides the storage area which is accessed by name. The user creates a storage area and uniquely identifies it by name within the Library. The UserStorage is provided for the user to store data that is plug-in specific: Capture and other plug-ins do not read/write or “understand” this data.

Once the UserStorage is created, the user may then retrieve an IStorage for that area at any time by requesting it by name. An IStorage is an independently readable/writeable directory within the library file. The user may create further independent streams within that storage or other storages in whatever manner meets his needs.

Class Hierarchy

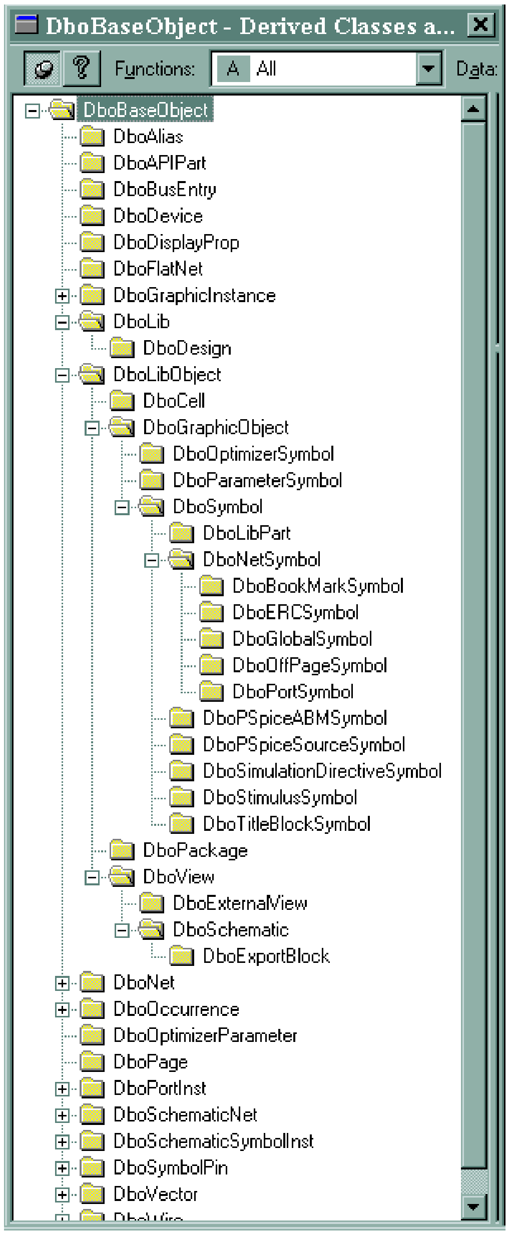


```
DboBaseObject
    DboAlias
    DboAPIPart
    DboBusEntry
    DboDevice
    DboDisplayProp
    DboFlatNet
    DboGraphicInstance
        DboGraphicArcInst
        DboGraphicBitMapInst
        DboGraphicBoxInst
        DboGraphicCommentTextInst
        DboGraphicEllipseInst
        DboGraphicLineInst
        DboGraphicPolygonInst
        DboGraphicPolylineInst
        DboGraphicSymbolVectorInst
        DboSymbolInstance
            DboNetSymbolInstance
                DboBookMark
                DboERC
                DboGlobal
                DboOffPageConnector
                DboPort
            DboOptimizerInstance
            DboParameterInstance
            DboPartInst
                DboDrawnInst
                DboPlacedInst
            DboPSpiceABMInstance
            DboPSpiceSimulationDirec
            DboPSpiceSourceInstance
            DboStimulusInstance
        DboTitleBlock
    DboLib
        DboDesign
    DboLibObject
        DboCell
        DboGraphicObject
            DboOptimizerSymbol
            DboParameterSymbol
            DboSymbol
```

```
            DboSymbol
                DboLibPart
                DboNetSymbol
                DboPSpiceABMSymbol
                DboPSpiceSourceSymbol
                DboSimulationDirectiveSymbol
                DboStimulusSymbol
                DboTitleBlockSymbol
            DboPackage
            DboView
                DboExternalView
                DboSchematic
                    DboExportBlock
        DboNet
            DboNetBus
            DboNetScalar
        DboOccurrence
            DboInstOccurrence
            DboNetOccurrence
            DboPortOccurrence
                DboPortBusMemberOccurrence
            DboTitleBlockOccurrence
        DboOptimizerParameter
        DboPage
        DboPortInst
            DboPortInst
            DboPortInst
                DboPort
        DboSchematicN
            DboSchema
            DboSchema
        DboSchematicS
            DboSchema
            DboSchema
            DboSchema
        DboSymbolPin
            DboSymbolI
            DboSymbolI
        DboVector
            DboArc
            DboBitMap
            DboBox
```

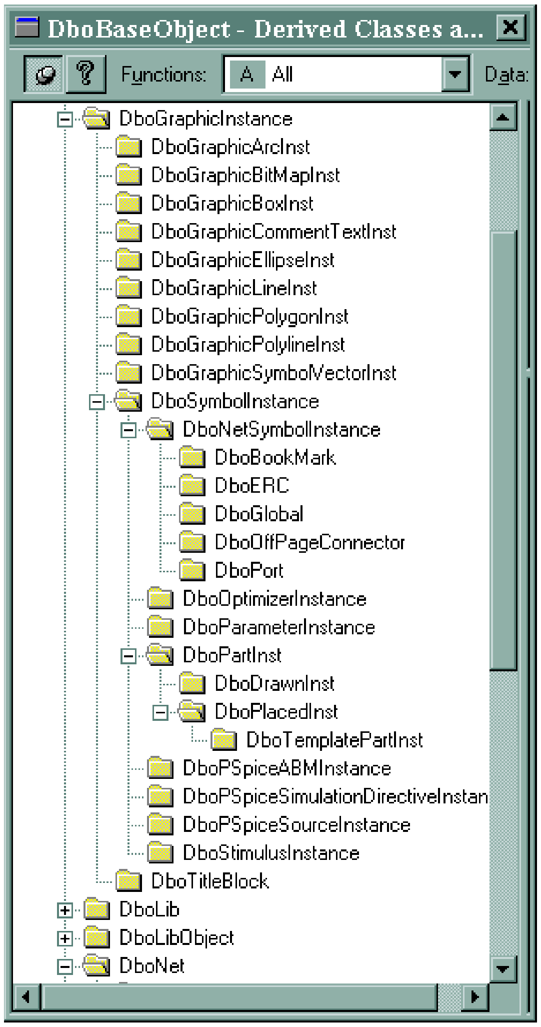
```
            DboSymbolPinScalar
        DboVector
            DboArc
            DboBitMap
            DboBox
            DboCommentText
            DboEllipse
            DboFill
            DboLine
            DboPolygon
            DboPolyline
            DboSymbolVector
        DboWire
            DboWire60
                DboWire60Bus
                DboWire60Scalar
            DboWireBus
            DboWireScalar
```

LibObject Class Hierarchy



```
DboBaseObject
    DboAlias
    DboAPIPart
    DboBusEntry
    DboDevice
    DboDisplayProp
    DboFlatNet
  + DboGraphicInstance
  - DboLib
        DboDesign
  - DboLibObject
        DboCell
      - DboGraphicObject
            DboOptimizerSymbol
            DboParameterSymbol
          - DboSymbol
                DboLibPart
              - DboNetSymbol
                    DboBookMarkSymbol
                    DboERCSymbol
                    DboGlobalSymbol
                    DboOffPageSymbol
                    DboPortSymbol
                DboPSpiceABMSymbol
                DboPSpiceSourceSymbol
                DboSimulationDirectiveSymbol
                DboStimulusSymbol
                DboTitleBlockSymbol
        DboPackage
      - DboView
            DboExternalView
          - DboSchematic
                DboExportBlock
  + DboNet
  + DboOccurrence
    DboOptimizerParameter
    DboPage
  + DboPortInst
  + DboSchematicNet
  + DboSchematicSymbolInst
  + DboSymbolPin
  + DboVector
```

Graphic Instance Hierarchy



```
DboGraphicInstance
    DboGraphicArcInst
    DboGraphicBitMapInst
    DboGraphicBoxInst
    DboGraphicCommentTextInst
    DboGraphicEllipseInst
    DboGraphicLineInst
    DboGraphicPolygonInst
    DboGraphicPolylineInst
    DboGraphicSymbolVectorInst
    DboSymbolInstance
        DboNetSymbolInstance
            DboBookMark
            DboERC
            DboGlobal
            DboOffPageConnector
            DboPort
        DboOptimizerInstance
        DboParameterInstance
        DboPartInst
            DboDrawnInst
            DboPlacedInst
                DboTemplatePartInst
        DboPSpiceABMInstance
        DboPSpiceSimulationDirectiveInstan
        DboPSpiceSourceInstance
        DboStimulusInstance
    DboTitleBlock
DboLib
DboLibObject
DboNet
```