SoftJin
Enabling Electronic Design

Oasis/Tutorial -- Tutorial for using OasisParser and OasisCreator
Version 0.2

Last modified: 29-Nov-2004 Mon 16:53

This tutorial explains how to use the Oasis library to read and write OASIS files. It assumes that you are familiar with the OASIS file format. The main classes you will need are OasisParser to read the file and OasisCreator to write the file.

You might find this tutorial easier to read if you first skimmed the following files.

oasis.h	for basic types used throughout the library
names.h	for OasisName and its subclasses
creator.h	for OasisCreator
parser.h	for OasisParser

Exceptions

OasisParser and *OasisCreator* methods may throw the following exceptions.

std::bad_alloc	memory allocation failed
std::overflow_error	integer overflow: some number is too big to represent
std::runtime_error	other unrecoverable error in parsing or writing

All these exceptions are derived from the standard class 'exception' and have a message that you can retrieve using *exception::what*().

A parser or creator object that throws an exception cannot be used further. That is, all exceptions are unrecoverable. If you catch an exception, do not call any other method of the object. All you can safely do is destroy the object and display the exception message.

SoftJin Infotech Pvt. Ltd
India: 227/70/A, Sigma Arcade, Marathahalli, Bangalore 560 037, India, Tel: +91-80-25234641, Fax: +91-80-25234643
Branch Off: 2900 Gordon Ave, Suite 100-11, Santa Clara, CA 95051, USA, Tel: (408) 773-1714, Fax: (408) 773-1745
Email: sales@softjin.com Web: www.softjin.com

Unlike many other libraries, this one does _not_ use exceptions to signal errors in the arguments. In particular, when you create a file with *OasisCreator* it is up to you to call the methods in the right order and pass them the right arguments. If you pass invalid arguments to *OasisCreator* methods, anything can happen, such as an assertion failure segmentation fault or just an invalid OASIS file.

OasisCreator

OasisCreator tries to make it easy to create OASIS files. In return for the ease of use it restricts your choices. Not every legal OASIS file can be created if you use OasisCreator. You cannot, for example, write name records with explicit reference-numbers because OasisCreator only writes the other form, with implicitly-assigned numbers. You cannot write PAD records.

To use *OasisCreator* you must *#include* its interface file, *creator.h*. The public methods of the class fall into five groups.

beginFile()	beginPlacement()
endFile()	beginText()
beginCell()	beginRectangle()
endCell()	beginPolygon()
	beginPath()
addFileProperty()	beginTrapezoid()
addCellProperty()	beginCircle()
addElementProperty()	beginXElement()
	beginXGeometry()
registerCellName()	
registerTextString()	setXYrelative()
registerPropName()	setCompression()
registerPropString()	
registerLayerName()	
registerXName()	

All except for *setXYrelative*() and *setCompression*() are virtual methods declared in *OasisBuilder*, which is a base class of *OasisCreator*. Later we explain why is significant.

To create a valid OASIS file, you must call these methods in a specific order and with correct arguments.

The first step is to construct the object. The constructor takes two arguments: the pathname of the OASIS file and a reference to an options struct. The only option is immediateNames. We describe it in the section Names below.

Then call beginFile() to open the file and write the magic string and START record.

```
void beginFile (const string& version,
                const Oreal& unit,
                Validation::Scheme valScheme)
```

The first argument is the OASIS version, which for now must be "1.0". The second argument is the unit (grid steps per micron) for the START record; this will be 1000 if your unit is 1 nm. (We explain the type *Oreal* below.)

The last argument is the validation scheme for the file, used to verify that the file has not been corrupted. This should be one of the following constants, all declared in *oasis.h*:

```
Validation::None
Validation::CRC32
Validation::Checksum32
```

CRC32 is better than *Checksum32* at detecting corrupted files, but if the file has large cells then (for complicated reasons) *OasisCreator* might not be able to compute the CRC while it is writing the file. It might have to reread the file, thus slowing it down and requiring the output file to be seekable. If validation is *None* or *Checksum32*, *OasisCreator* writes the file sequentially. Then the output may be a pipe.

To finish writing the OASIS file, call *endFile*(). This writes all the name tables and the END record, and closes the file. Do not call any other method after *endFile*().

Here is an example of the code:

```
OasisCreator creator("foo/bar.oasis");
creator.beginFile("1.0", 1000, Validation::Checksum32);
// ... code to create all the cells
creator.endFile();
```

To write file-level properties like S_TOP_CELL, construct a Property object and call *addFileProperty*() for each property immediately after *beginFile*(). The section `Properties' below explains how to construct Property objects.

Strings and Reals

The library uses the standard class *std::string* to pass string fields and the class *Oreal* to pass OASIS fields that are real numbers. The string class provides a conversion from a C string. That is why you can supply "1.0" directly as the value of the version argument of *beginFile*().

Oreal lets you control to some extent the storage representation of the number. For example,

Oreal(137) creates a real number that will be written as a type-0 real (a positive integer);

Oreal(-3, 5) will be written as a type-5 real (a negative rational); and

Oreal(3.1415) will be written as a type-7 real (a double-precision float).

Oreal provides implicit conversions from integers and floating-point numbers. So you can provide an integer, float, or double value to a function taking a const *Oreal&* argument.

When reading an OASIS file you can use *Oreal* methods like *isInteger*() and *isRational*() to get the type of the real, or just use *getValue*() if you don't care about the representation.

Cells

To write a cell, first call *beginCell*(), then create the elements, and finally call *endCell*(). To write cell-level properties, construct a Property object and call *addCellProperty*() for each cell property immediately after *beginCell*(), before creating the elements.

To write an element record, call the appropriate begin method for that element, e.g., *beginPolygon*() for polygons. If the element has properties, call *addElementProperty*() for each property of the element. Although *OasisCreator* has an *endElement*() method -- it inherits the empty method from *OasisBuilder* -- there is no need to call it.

The first argument of *beginPlacement*(), which writes a PLACEMENT record, is a pointer to a *CellName*. If you registered this *CellName* before calling *beginPlacement*(),

OasisCreator writes the reference-number form of the record. Otherwise it writes the *cellname-string* form. See the section `Names' below.

One of the arguments of *beginText*(), which writes a TEXT record, is a pointer to a *TextString*. As with *beginPlacement*(), *OasisCreator* writes the reference-number form of the record if you registered the TextString earlier. You should retain the *TextString* object until *endFile*() returns, even if you did not register it.

OasisCreator uses modal variables whenever possible when it writes the element and PROPERTY records. If you are concerned about the size of the OASIS file, sort the elements before writing them, to let *OasisCreator* use the modal variables as much as possible. For instance, write all elements in a given layer together. Or if there are many rectangles of a given width, write them together. If an element has a Rep_Arbitrary or Rep_GridArbitrary repetition with lots of points, sort the points before adding them to the repetition.

Absolute and relative xy-mode

Before each element you can call *setXYrelative*() to change xy-mode from absolute to relative or back. A true argument sets xy-mode to relative and a false argument sets it to absolute. Each cell begins in absolute mode.

In absolute mode *OasisCreator* writes the X and Y coordinates of each element. In relative mode it writes the delta between the coordinates supplied and those of the last similar element written. If you write nearby elements together then the deltas will usually be smaller than the actual coordinates. Since integers in the OASIS file have a variable size, this tends to make the file smaller.

With relative coordinates, however, it is up to you to ensure that computing the delta will not result in integer overflow. If you write one element at $(-2^{30}, 0)$ and the next at $(2^{30}, 0)$, the X delta is 2^{31}, which *OasisCreator* cannot represent on a 32-bit machine. On such machines it will throw the exception *overflow_error*. Since all *OasisCreator's* exceptions are unrecoverable, you will then have to abort writing the file.

On a 64-bit system you need not worry about integer overflow, but on a 32-bit system you should probably remain in absolute mode to be safe. If you really want to use relative mode you can do what *GdsToOasisConverter* in *../conv/gds-oasis.cc* does: check each delta, switch to absolute mode if it would overflow, write the element, then switch back to relative mode.

Repetition

The last argument of most of the element-writing methods is a pointer to a Repetition object. Pass Null if there is no repetition. Otherwise initialize a Repetition object using *makeMatrix*() or one of the similar functions.

For the non-uniform repetitions you also have to add the offsets or deltas. When creating a Repetition of type *Rep_VaryingX*, *Rep_VaryingY*, or their gridded variants, add 0 as the first offset. Similarly when creating a *Rep_Arbitrary* or *Rep_GridArbitrary*, add (0,0) as the first delta. The first point must have zero offset because all offsets in the Repetition are with respect to the first point.

Below is an example of using a Repetition. Note that the first argument of *makeGridVaryingX*(), the number of points in the repetition, is only advisory.

```
Repetition rep;
rep.makeGridVaryingX(10, 1000);
rep.addVaryingXoffset(0);
rep.addVaryingXoffset(2000);
rep.addVaryingXoffset(5000);
creator.beginRectangle(3, 0, 1000, 1000, 2000, 3000, &rep);
```

The offsets must be in non-decreasing order. In other words, you must list the points from left to right. All offsets for gridded repetitions must be a multiple of the grid.

As with all the other fields, before *OasisParser* writes a repetition it checks if it is identical to the last repetition it wrote. If so, it writes a type-0 (Rep_ReusePrevious) repetition instead.

PointList

The element functions *beginPolygon*() and *beginPath*() take a *PointList* argument for the sequence of points. To initialize a *PointList*, invoke its *init*() method with the type of point-list as argument (see Table 7 in the OASIS specification), and then add the points one by one. Each point in the *PointList* is the delta between it and the first point. Hence the first point must be (0,0). The coordinates of the points you add must be consistent with the type. For example, the following is illegal:

```
PointList  ptlist;
ptlist.init(ManhattanHorizFirst);
ptlist.addPoint(Delta(0, 0));
ptlist.addPoint(Delta(3, 3));
```

This is illegal because for ManhattanHorizFirst the second point must have the form (x,0) for some x. addPoint does not verify that the points added meet the constraints imposed by the point-list type.

Compression

OasisCreator by default compresses the output. That is, it puts the contents of each cell and name table into a CBLOCK record. If for some reason you don't want this, call *setCompression*(false). Later if you want to resume compression, call *setCompression*(true). You may call the function anytime; it will take effect from the next cell or name table written. *OasisCreator* does not provide a way to compress part of a cell or name table. It would be trivial to provide it, but there seems no need.

Names

OASIS has six types of name records: CELLNAME, TEXTSTRING, PROPNAME, PROPSTRING, LAYERNAME and XNAME. The library stores the contents of these records in the correspondingly-named classes *CellName*, *TextString*, *PropName*, *PropString*, *LayerName*, and *XName*. The first four are just *typedefs* for *OasisName*, which is the base class of the other two.

Each *OasisName* has a name, which you can access using its *getName*() method, and a list of properties, which may be empty. For *TextString* and *PropString* this `name' is just a string. Although the names that appear in the name records of OASIS files have associated reference numbers, the library hides these from the application, except for XNAMEs. Because the XNAME reference numbers may be used in XELEMENT and XGEOMETRY records, which are opaque to the library, they are made available to the application in the *XName* class. *OasisCreator* expects the application to set those numbers and ensure that they are unique.

OasisCreator provides a registration function for each type of name, e.g., *registerCellName*() for *CellNames*. When you register a name object, *OasisCreator* assigns it a reference-number and keeps a pointer to it. If it later writes a record that refers to the name it writes the reference-number instead of the name.

If immediateNames was true in the constructor's options argument, OasisCreator writes the name record immediately in the registration function itself. You must therefore call the name-registration functions only when it is legal to have a name record. For example, the sequence of calls

```
registerTextString()
beginText()
registerTextString()
beginText()
```

would result in an invalid OASIS file because each call to registerTextString() would write a TEXTSTRING record, and one or both of the TEXTSTRING records would appear within a cell.

If immediateNames was false in the constructor's options argument, OasisCreator writes all registered names at the end of the file, when you call endFile(). It groups records of each type to form a name table.

The option immediateNames was added because some OASIS tools require each name record to appear in the file before it is used. You should set it to true only if you want the OASIS file readable by such tools.

The name table for a given type of name (e.g., cell name) will be strict if and only if you registered _every_ name of that type before using it. When a name table is strict, *OasisParser* can read all the name records for that type of name by jumping directly to the start of the table. If any name table is not strict, *OasisParser* scans the whole file twice, the first time just to collect the name records. It is therefore a good idea to ensure that all name tables are strict by registering each name before using it.

OasisCreator remembers the starting offset of each cell that it writes. If you register a *CellName* that you pass to *beginCell*(), either before or after *beginCell*(), then when OasisCreator gets around to writing the cellname table it will add an S_CELL_OFFSET property to the CELLNAME record. Thus by registering the names of all cells you will create a file that allows random access to the cells.

When registering names, ensure that *CellNames*, *TextStrings*, and *PropNames* are unique. Do not register any name more than once. You must retain every name object until *endFile*() returns. That more or less forces you to heap-allocate them.

Assume that you have a class member called *cellNames* declared like this.

```
PointerVector<CellName>  cellNames;
```

The template class *PointerVector*, defined in *../misc/ptrvec.h*, is a vector of pointers. The vector owns all the objects it points to, i.e., the *PointerVector* destructor deletes them. To create a *CellName* in an exception-safe way, you will some roundabout code. The obvious approach has a possible memory leak.

```
CellName*  cellName = new CellName("foo");
cellNames.push_back(cellName);
```

The trouble with this is that if *push_back*() threw an exception the memory allocated to the *CellName* object would be lost. Instead you have to do something like this, using *auto_ptr* to delete the *CellName* if *push_back* throws.

```
auto_ptr<CellName>  acn(new CellName("foo"));
cellNames.push_back(acn.get());
CellName*  cellName = acn.release();
```

Now you can register the cell and use it:

```
creator.registerCellName(cellName);
creator.beginCell(cellName);
```

You will probably need a map from strings to *CellNames* to avoid creating two *CellName* objects with the same name; similarly for the other types of names. See class *GdsToOasisConverter* in *../conv/gds-oasis.cc* for an example.

Properties

A PROPERTY record in OASIS has a name and a sequence of values. The name may be specified directly as a string or as a reference to a PROPNAME record which has the name string. In this library, class Property stores the contents of PROPERTY records.

The Property constructor has this prototype:

```
Property (PropName* name, bool isStd)
```

The first parameter gives the name of the property. You must always supply a *PropName**, even if you want the PROPERTY record to contain a *propname-string* rather than a reference-number. When *OasisCreator* writes the PROPERTY record it uses the reference-number form if the PropName object was registered earlier using *registerPropName*(); otherwise it will use the *propname-string* form.

The second parameter, isStd, must be true if you are creating one of the standard properties described in the appendix of the OASIS specification, and false otherwise.

The *Property* object does _not_ take ownership of the *PropName* object. You remain responsible for deleting it, which of course you must do only after deleting the Property object itself. Creating a Property object in an exception-safe way is even trickier than

creating an *OasisName*. Assume that you have a class member call *propNames* declared like this.

```
PointerVector<PropName>  propNames;
```

First we create the *PropName* object, add it to *propNames* to give it an owner, and register it with the *OasisCreator*:

```
auto_ptr<PropName>  apn(new PropName("S_TOP_CELL"));
propNames.push_back(apn.get());
PropName*  propName = apn.release();
creator.registerPropName(propName);
```

Now we create the Property object, which itself should have an owner.

```
auto_ptr<Property>  prop(new Property(propName, true));
```

The final step is to add the values to the property, using the overloaded *addValue*():

```
prop->addValue(PV_Name_String, "root");
```

The C string literal "root" is implicitly converted to a C++ string. If you add a *PropString* value, bear in mind that *PropValue* does _not_ take ownership of the *PropString*. Ensure that the *PropString* has an owner by storing a pointer to it in a *PointerVector* or *auto_ptr*.

Use *OasisName::addProperty*() if you want to add this property to a name. If you do it, you must release the pointer from the *auto_ptr* because *OasisName* takes ownership of the Property object. You still remain the owner of the *PropName*, though. (Yes, all this ownership stuff is confusing. No, *boost::shared_ptr* is not the answer.) The general rule is that whoever creates a name object remains responsible for it.

Because *OasisCreator* writes the name records, with their associated PROPERTY records, only when you call *endFile*(), you can add properties to a name anytime before then.

To add an element property, construct a Property object and call *addElementProperty*() to write the PROPERTY record. The Property object is no longer needed once *addElementProperty*() returns. You can destroy it, or you can change the property values using *PropValue::assign*() and reuse the property for another element. Similarly *addFileProperty*() and *addCellProperty*() write the PROPERTY records immediately, so you can destroy or reuse the Property objects.

OasisParser

Let us turn now to the task of reading OASIS files using *OasisParser*. This class is the highest-level of the three classes for reading OASIS files. Below it is *OasisRecordReader*, and at the bottom is *OasisScanner*. *OasisParser* hides some of the information in the input file, e.g., the reference-numbers in all name records apart from XNAME, and the presence of XYABSOLUTE and XYRELATIVE records. If you want this information you should use *OasisRecordReader*, but first see the next section, which explains why you probably should _not_ use *OasisRecordReader* or its corresponding output class *OasisRecordWriter*.

OasisParser is an event-based parser. That means that instead of building a tree from the input data it generates events, i.e., makes callbacks to application code, as it recognizes parts of the input file. The callbacks are methods of the class *OasisBuilder*. To use the parser you must derive a class from *OasisBuilder*, override the virtual methods, and pass an instance to *OasisParser::parseFile*().

Your code will look something like this:

```
#include "oasis/parser.h"
#include "oasis/builder.h"
using namespace Oasis;

class MyBuilder : public OasisBuilder {
    // override all virtual methods
    virtual void beginFile();
    virtual void endFile();
    // ...
};

void DisplayWarning(const char* msg) {
    printf(stderr, "%s\n", msg);
}

void ProcessFile (const char* path) {
    OasisParser  parser(path, DisplayWarning);
    MyBuilder  builder;
    parser.parseFile(&builder);
    // ...
}
```

The code to pass a non-static class method as the *WarningHandler* argument is complicated. Supposing the warning handler is *MyBuilder::showWarning*(), the code would be like this:

```
#include <functional>
using namespace std;

MyBuilder   builder;
OasisParser parser(path,
                bind1st(mem_fun(&MyBuilder::showWarning),
                    &builder));
```

OasisParser makes a preliminary pass of the input file to collect all the name records. If **all** name tables are strict, it does this by jumping directly to each name table. Otherwise it scans the whole file sequentially. Either way, the input file must be seekable. That implies that you cannot pipe the output of another command to *OasisParser*.

OasisParser first invokes the builder's *beginFile*() method. Then, for each name that appears in a name record or CELL record, it invokes the appropriate registration method, e.g., *registerCellName*() for cell names. Next, if there are any file-level properties, OasisParser invokes *addFileProperty*() for each. Next, if there are any cells, *OasisParser* invokes the cell methods (described below) for each. Finally, at the end of the file, it invokes *endFile*().

This sequence of calls can be described more concisely in Extended BNF like this:

```
<file> ::= beginFile()
             { <name> }*
             { addFileProperty() }*
             { <cell> }*
           endFile()

<name> ::= registerCellName()
         | registerTextString()
         | registerPropName()
         | registerPropString()
         | registerLayerName()
         | registerXName()
```

The sequence of calls for <*cell*> follows a similar pattern. First *beginCell*(), then *addCellProperty*() for each cell-level property (if any), then the element calls, and finally *endCell*(). And again for elements, the lowest level of the hierarchy.

```
<cell> ::= beginCell()
             { addCellProperty() }*
             { <element> }*
           endCell()

<element> ::= <beginElem>
                { addElementProperty() }*
              endElement()

<beginElem> ::= beginPlacement()
              | beginText()
              | beginRectangle()
              | beginPolygon()
              | beginPath()
              | beginTrapezoid()
              | beginCircle()
              | beginXElement()
              | beginXGeometry()
```

OasisParser retains all the name objects it passes to the registration methods as long as it lives. You should not modify or destroy those names. The Property objects attached to those names are also permanent. But the Property objects passed to *addFileProperty*(), *addCellProperty*() and *addElementProperty*() are transient: *OasisParser* destroys the objects as soon as the method returns.

OasisParser::parseFile() parses an entire OASIS file. If you want to parse a single cell, call parseCell(). You need to know the name of the cell. Unfortunately, OasisParser does not for now provide a way to get the names of all cells in an OASIS file. You can call parseCell() repeatedly, for the same cell or different cells.

OasisParser and OasisCreator

Recall that *OasisCreator* is derived from *OasisBuilder*. If you reread the paragraphs that describe the legal ordering for calls to *OasisCreator* methods, you will see that the order in which *OasisParser* calls the *OasisBuilder* methods meets all the *OasisCreator* requirements. (That should not be a surprise.) You can therefore pass an instance of *OasisCreator* as the builder argument to *OasisParser::parseFile*(). This will result in a logical copy of the input file.

The copy is only logical, not physical. That is, the output file will have the same semantics as the input file, but the bytes may be different. For instance, all coordinates will be absolute (recall that *OasisParser* swallows XYABSOLUTE and XYRELATIVE

records), name records might be in a different order, and the output file will use CBLOCK records even if the input file did not.

OasisRecordReader and OasisRecordWriter

OasisRecordReader and *OasisRecordWriter* read and write raw OASIS records. `Raw' means that the record structures that are passed out of or into those classes contain only what is in the file. For instance, *OasisRecordReader* does not create *OasisName* objects or resolve reference-numbers into names; nor does it use modal variables to fill in fields omitted from element and PROPERTY records.

The classes are therefore less useful than they might seem at first sight. Consider the problem of writing a program oasis-filter whose arguments are the name of an OASIS file and a list of layer numbers. The program should write a modified file that contains only elements in the listed layers. *OasisRecordReader* and *OasisRecordWriter* seem ideal for this. All we have to do is read each record and write it unless it is an element record whose layer is not in the selected set. We must also skip PROPERTY records belonging to skipped element records. This looks straightforward.

But there are two problems with this. The minor one is that element records may omit the layer number, so we must keep track of the modal variable ourselves. The major problem is that file offsets may appear as data in the file. A CELLNAME record may have a property S_CELL_OFFSET whose value is the file offset of the cell. If this offset is different in the output file because of the dropped records, we must update it. But how?

If the CELLNAME record appears before the CELL record, we need two passes of the input file -- the first to collect all the cell offsets and the second to write the correct values. But wait, it gets worse. Integers in OASIS have a variable size, and neither *OasisRecordWriter* nor *OasisWriter* (the bottom layer of code, which actually writes the integers) provides a way to specify how many bytes an integer should occupy in the file. So merely writing different values in the PROPERTY record in the second pass might change the offsets of CELL records further on.

Even recognizing an S_CELL_OFFSET property is hard. Remember that PROPERTY records may use reference-numbers to identify the property name, and the PROPNAME record identified by the reference-number may appear later in the file. Since the reference-numbers may be assigned implicitly, you might have to keep track of all the PROPNAME records you see.

The conclusion: Don't use *OasisRecordReader* or *OasisRecordWriter*. Or, for that matter, *AsciiRecordReader* and *AsciiRecordWriter*, which are similar classes that deal with the ASCII version of OASIS files. (*sample.ascoas* describes the format of these ASCII files.) These classes exist mainly for *oasis2ascii* and *ascii2oasis*.