Chapter 3.
Object Libraries

Topic: *Building Applications

The source code for a component is built into object libraries for an application to link against. A component may have more than one library associated with it. The ACIS libraries may be grouped into two main categories:

Core Libraries . . . . Provide ACIS modeling functionality. Most components have at least one core library, but some components are for interface support only and do not have a core library.

Interface/Support Libraries . . . . Do *not* provide ACIS modeling functionality, but are simply for supporting an interface into ACIS, such as Scheme.

The core ACIS libraries are available as either static link libraries or shared (dynamic link) libraries. A *static library* means that the library's referenced object code becomes part of the application's executable. A *shared library* means that the library's object code does *not* become part of the application's executable (so the executable is smaller), but is loaded into memory at run-time instead.

Note *A shared library is called a* Dynamic Link Library *(DLL) on some systems (e.g., Windows). When referring to a shared library in a non-platform-specific context, the* Spatial *documentation may use the term* shared/DLL.

The top-level ACIS install directory contains a directory for each component of ACIS. No source code is allowed in the top-level or component directories. All source code is located in subdirectories of the component directories. Each component directory contains one subdirectory for each library that is shipped as part of the component. Some component directories may also contain other subdirectories that are not built into libraries.

Libraries

Topic: *Building Applications

This section describes the ACIS libraries. The libraries are grouped into the following tables:

Table	Title	Description
3-1	ACIS Core Libraries	Core libraries for **ACIS 3D Geometric Modeler**. This does not include components for specific renderers.
3-2	Renderer Component Core Libraries	Core libraries specific to the renderer components. *These libraries are mutually exclusive*. This means that an application can only link in one renderer component library.
3-3	Scheme Libraries	Libraries for the Scheme interface (including Scheme AIDE application).

For each table, the columns contain the following information:

Library . . . . . . The library's "unadorned" filename. This generally corresponds to the name of the component subdirectory from which the library was built. Any library whose unadorned filename ends in _scm contains Scheme extensions.

For Windows, the actual static library filename will be <*library*>.lib, where <*library*> is the unadorned library filename. For DLLs, the library filename does not use <*library*>, but rather the module identifier string (as described in the *Type* column), plus the current release number. The DLL filename is <*MODULE*><*A3DT_MAJOR*><*A3DT_MINOR*>.DLL, where <*MODULE*> is the module identifier string, and <*A3DT_MAJOR*> and <*A3DT_MINOR*> are environment variables that specify the current major and minor release numbers. For example, blend.lib would be the core static object library for the Blending Component, and BLND70.DLL would be the core DLL for the Blending Component for Release 7.0.

For UNIX systems, the actual library filename will be lib<*library*>.a for static libraries, and lib<*library*>.so for shared libraries on most UNIX systems, and lib<*library*>.sl for shared libraries on HP systems. For example, libblend.sl would be the core object library file for the Blending Component if built as shared on HP.

For Macintosh systems, the actual library name is <*library*>.lib for component static libraries. For shared libraries, there are only two libraries, AcisCore.shlb and AcisModules.shlb. AcisCore.shlb contains the code for the Kernel Component; AcisModules.shlb contains the code for the other core ACIS components. There is one static library, scmext.lib, that contains all the code from the *_scm subdirectories as well as the contents of the scm_ext subdirectory.

Type . . . . The type of library, either static or shared/DLL.

If the type is *Static*, the library is available only as a static library. If the type is *Shared/DLL*, the library may be built as either static or shared/DLL, and the second line in this column contains the module identifier string used in library macros (e.g., DECL_<*MODULE*> macro that controls the import/export of symbols from the shared library). Refer to section *Library Macros* for more information.

Description . . . A brief description of the library.

The first line is the library's component. This includes the component's top-level directory in parentheses. The second line is the identifier string, if any, that is used in initialization and termination routines for this library (refer to section *Initialization and Termination*). Any additional comments are contained on the following lines.

Table 3-1. ACIS Core Libraries

Library	Type	Description
abl_husk	Shared/DLL ABL	Advanced Blending Component (abl) advanced_blending
admgi_control	Shared/DLL DMGI_CONTROL	ACIS Deformable Modeling Graphic Interaction Component (admgi) admgi_control Replaceable library to register objects with GI rendering
admgi_draweng	Shared/DLL DMGI_DRAWENG	ACIS Deformable Modeling Graphic Interaction Component (admgi) admgi_draweng Replaceable library translating draw requests into GI primitives
admhusk	Shared/DLL ADM	ACIS Deformable Modeling Component (adm) deformable_modeling
admicon	Shared/DLL ADM_ICON	ACIS Deformable Modeling Component (adm) admicon Non–required library; replacements for surface icons

Library	Type	Description
phlv5_husk	Shared/DLL PHLV5	Precise Hidden Line Removal V5 Component (phlv5) hidden_line_removal
apfill	Shared/DLL APFILL	PowerFill Component (apfill) powerfill
baseutil	Shared/DLL BASE	Base Component (base) base
blend	Shared/DLL BLND	Blending Component (blnd) blending
boolean	Shared/DLL BOOL	Boolean Component (bool) booleans
cathusk	Shared/DLL CAT	CATIA Translator Component (catia) catia
clear	Shared/DLL CLR	Clearance Component (clr) clearance
constrct	Shared/DLL CSTR	Constructors Component (cstr) constructors
cover	Shared/DLL COVR	Covering Component (covr) covering
ct_husk	Shared/DLL CT	Cellular Topology Component (ct) cellular_topology
dmicon	Shared/DLL DM_ICON	Standalone Deformable Modeling Component (ds) dmicon Replaceable library with default icons
dshusk	Shared/DLL DM	Standalone Deformable Modeling Component (ds) sdmhusk
euler	Shared/DLL EULR	Euler Operations Component (eulr) euler_ops
faceter	Shared/DLL FCT	Faceter Component (fct) faceter
ga_husk	Shared/DLL GA	Generic Attributes Component (ga) generic_attributes
gihusk	Shared/DLL GI	Graphic Interaction Component (gi) graphic_interaction
healhusk	Shared/DLL HEAL	Healing Component (heal) healing

Library	Type	Description
ihl_husk	Shared/DLL IHL	Interactive Hidden Line Component (ihl) interactive_hidden_line
intersct	Shared/DLL INTR	Intersectors Component (intr) intersectors
kernel	Shared/DLL KERN	Kernel Component (kern) kernel
lawutil	Shared/DLL LAW	Laws Component (law) law
lop_husk	Shared/DLL LOP	Local Operations Component (lop) local_ops
lopt_husk	Shared/DLL LOPT	Local Operation Tools Component (lopt) lopt_ops
offset	Shared/DLL OFST	Offsetting Component (ofst) offsetting
operator	Shared/DLL OPER	Operators Component (oper) operators
part	Shared/DLL PART	Part Management Component (part) part_manager
phl_husk	Shared/DLL PHL	Precise Hidden Line Component (phl) precise_hidden_line
pid_husk	Shared/DLL PID	Persistent ID Component (pid) persistent_id
proehusk	Shared/DLL PROE	Pro/E Translator Component (proe) proe
rbi_husk	Shared/DLL RBI	Repair Body Intersections Component (rbi) rbi
rem_husk	Shared/DLL REM	Remove Faces Component (rem) face_removal
rnd_husk	Shared/DLL RB	Rendering Base Component (rbase) rendering
sbool	Shared/DLL SBOOL	Selective Booleans Component (sbool) sbooleans
shl_husk	Shared/DLL SHL	Shelling Component (shl) shelling

Library	Type	Description
skin	Shared/DLL SKIN	Advanced Surfacing Component (skin) skinning
stephusk	Shared/DLL STEP	STEP Translator Component (step) step
stchhusk	Shared/DLL STEP	Stitch Component (stitch) stitching
sweep	Shared/DLL SWP	Sweeping Component (swp) sweeping
transutl	Shared/DLL TRANS	Translator Utility Component (trans) No identifier
vdahusk	Shared/DLL VDA	VDA–FS Translator Component (vda) vda
warphusk	Shared/DLL WARP	Space Warping Component (warp) warp
visman	Shared/DLL VM	Visualization Manager Component (vm) No identifier
xgeometric	Static	Translation Geometry Component (xgeom)

Table 3-2. Renderer Component Core Libraries

Library	Type	Description
br_husk	Shared/DLL BR	Basic Rendering Component (br) basic_rendering
gl_husk	Shared/DLL GL	OpenGL Rendering Component (gl) opengl_rendering – Select platforms only

Table 3-3. Scheme Libraries

Library	Type	Description
abl_scm	Static	Advanced Blending Component (abl) advanced_blending_scmext
phlv5_scm	Static	Precise Hidden Line Removal V5 Component (phlv5) hidden_line_removal_scmext
apfill_scm	Static	PowerFill Component (apfill) powerfill_scmext

Library	Type	Description
blnd_scm	Static	Blending Component (blnd) blending_scmext
bool_scm	Static	Boolean Component (bool) booleans_scmext
cat_scm	Static	CATIA Translator Component (catia) catia_scmext
covr_scm	Static	Covering Component (covr) covering_scmext
cstr_scm	Static	Constructors Component (cstr) constructors_scmext
ct_scm	Static	Cellular Topology Component (ct) cellular_topology_scmext
dfct_scm	Static	Faceter Component (fct) display_facets_scmext
ds_scm	Static	ACIS Deformable Modeling Component (adm) deformable_modeling_scmext
eulr_scm	Static	Euler Operations Component (eulr) euler_ops_scmext
fct_scm	Static	Faceter Component (fct) faceter_scmext
ga_scm	Static	Generic Attributes Component (ga) generic_attributes_scmext
gi_scm	Static	Graphic Interaction Component (gi) graphic_interaction_scmext
gl_scm	Static	OpenGL Rendering Component (gl) opengl_rendering_scmext – Select platforms only
heal_scm	Static	Healing Component (heal) healing_scmext
iges_scm	Static	IGES Translator Component (iges) iges_scmext
igl_scm	Static	Interactive OpenGL Component (igl) interactive_opengl_scmext
ihl_scm	Static	Interactive Hidden Line Component (ihl) interactive_hidden_line_scmext

Library	Type	Description
intr_scm	Static	Intersectors Component (intr) intersectors_scmext
kern_scm	Static	Kernel Component (kern) kernel_scmext
lop_scm	Static	Local Operations Component (lop) local_ops_scmext
main	Static	Scheme Support Component (scm) No identifier – Command window and event processing
ofst_scm	Static	Offsetting Component (ofst) offsetting_scmext
oper_scm	Static	Operators Component (oper) operators_scmext
parted	Static	Scheme Support Component (scm) No identifier
phl_scm	Static	Precise Hidden Line Component (phl) precise_hidden_line_scmext
pmhusk	Static	Scheme Support Component (scm) No identifier – Scheme AIDE application interface to part management and graphic interaction
proe_scm	Static	Pro/E Translator Component (proe) proe_scmext
rbi_scm	Static	Repair Body Intersections Component (rbi) rbi_scmext
rem_scm	Static	Remove Faces Component (rem) face_removal_scmext
render	Static	Scheme Support Component (scm) No identifier – Scheme AIDE application rendering interface
rnd_scm	Static	Rendering Base Component (rbase) rendering_scmext
sbool_scm	Static	Selective Booleans Component (sbool) sbooleans_scmext

Library	Type	Description
scheme	Static	Scheme Support Component (scm) No identifier – Scheme Interpreter
scmext	Static	Scheme Support Component (scm) scmext – Basic Scheme extensions
shl_scm	Static	Shelling Component (shl) shelling_scmext
skin_scm	Static	Advanced Surfacing Component (skin) skinning_scmext
step_scm	Static	STEP Translator Component (step) step_scmext
stch_scm	Static	Stitch Component (stitch) stitching_scmext
swp_scm	Static	Sweeping Component (swp) sweeping_scmext
testext	Static	Scheme Support Component (scm) No identifier
vda_scm	Static	VDA–FS Translator Component (vda) vda_scmext
warp_scm	Static	Space Warping Component (warp) warp_scmext

Library Initialization and Termination

Topic: *Building Applications, *Modeler Control

The libraries for each ACIS component must be initialized before use and terminated after use. Initialization and termination functions are provided for each component library. API functions api_initialize_<*ident*> should be called after api_start_modeller and API functions api_terminate_<*ident*> should be called before api_stop_modeller.

When an ACIS component is initialized, it initializes any components upon which it depends, so an application only needs to initialize the highest level components in the flow of dependency that it uses. It is the developer's responsibility to make sure any component is initialized by the application before use.

Note *The string* <ident> *should be substituted with the library identifier listed in the* Description *column of the appropriate library table in section* Libraries.

The library initialization for the Base Component, Kernel Component, and Laws Component is built into function api_start_modeller, and their termination is built into api_stop_modeller, so an application may not need to explicitly initialize and terminate these components.

Library Dependencies

Topic: *Building Applications

A library may depend on one or more other libraries. An application must link in all libraries on which any referenced library depends.

Refer to the *Architecture* chapter of the *3D ACIS Getting Started Guide* for a graph that shows the dependencies between the core components. This graph indicates the standard dependencies between components; occasionally, an optional dependency between two components may arise if some optional arguments to a function, method, etc. are used. Additional dependencies may be introduced by callbacks.

A component library can only use items (e.g., functions, classes, etc.) defined in component libraries on which it depends—those that can be reached by following the dependency path in the graph.

Library Macros

Topic: *Building Applications

Macros are used to help set up libraries for use as shared/DLLs.

Note *The string* <MODULE> *should be substituted with the shared library module identifier listed in the* Type *column of the appropriate library table in section* Libraries. *The string* <module> *should be replaced with the lowercase version of this.*

A dcl_<*module*>.h header file exists for each component (module) that is to be available as a DLL. This header file defines the macro DECL_<*MODULE*> based on the settings of ACIS_DLL and EXPORT_<*MODULE*>. The DECL_<*MODULE*> macro is used to indicate whether a symbol is being exported or imported from a DLL.

A module name argument exists for each of the following macros (use NONE as the module name argument if the symbol will not be in a DLL):

ATTRIB_FUNCTIONS
LIST
MODULE_DEF
DISPATCH_DECL
MASTER_ATTRIB_DECL
MODULE_REF
ENTITY_FUNCTIONS
MASTER_ENTITY_DECL

For example, in Release 2.1, if the MODULE_DEF macro was used as:

```
MODULE_DEF(”api”)
```

it must be changed for Release 3.0 to:

```
MODULE_DEF(”api”, KERN)
```

The DECL_<*MODULE*> macro or a module argument to the macros listed above must be used to allow a function or global variable defined in a lower library to be used by higher libraries and/or applications.

THIS_LIB and PARENT_LIB macro definitions are now required wherever THIS() and PARENT() macro definitions are required for entity declarations and implementation. For example, the following definition for Release 2.1:

```
#define THIS() REFINEMENT
#define PARENT() ENTITY
```

must be changed for Release R10 to:

```
#define THIS() REFINEMENT
#define THIS_LIB FCT
#define PARENT() ENTITY
#define PARENT_LIB KERN
```

Using Shared Libraries (DLLs)

Topic: *Building Applications

The advantages of shared/DLL libraries include:

- Storage space is reduced
- Memory *may* be used more efficiently by the operating system at run-time

In general, an application can use either static or shared/DLL libraries without modification to its source code. However, when using ACIS DLLs, precautions must be taken when compiling your application to prevent errors. Also, if you need to create your own custom ACIS component as a shared library, the component must meet certain requirements in order to be built as a shared ACIS library.

DLLs

Topic: *Building Applications

If you are going to link your Windows NT application against the ACIS DLLs, you must define the symbol ACIS_DLL when you compile any file that references global symbols defined in any ACIS DLL to avoid unresolved symbol errors. Also, you must use compiler flags consistent with those used to build the DLLs to avoid run-time problems.

Ensure that you are using the same version of the C Runtime Library when you compile and link your application as was used when the DLLs were built. Also, ensure that the same C Runtime Library was used to build *all* of the DLLs being linked to the application. Otherwise, you will get access violation errors when working with pointers or file handles passed between DLLs or between the application and the DLLs.

Building ACIS Shared Libraries

Topic: *Building Applications

When adding your own custom components to ACIS, you may want the libraries to be available as shared/DLL. When using shared/DLL libraries, it is important to keep the order of libraries in mind. Functions from higher libraries can not be called from lower libraries.

The following steps should be done for each library that is to be built as a shared ACIS library. An example (for Constructors or Intersectors libraries) is shown for most steps.

Note *The string* <MODULE> *should be substituted with the shared library module identifier listed in the* Type *column of the appropriate library table in section* Libraries. *The string* <module> *should be replaced with the lowercase version of this.*

1. Create a dcl_<*module*>.h file in the library's directory.

   This header file defines the DECL_<*MODULE*> macro to indicate whether symbols are being exported or imported from the shared library. It also causes the library to be searched automatically by any file that includes this header on Windows systems.

```
#ifndef DECL_CSTR
#ifdef ACIS_DLL
# ifdef EXPORT_CSTR
#   define DECL_CSTR __declspec(dllexport)
# else
#   define DECL_CSTR __declspec(dllimport)
#   ifdef NT
#       pragma comment(lib, ”constrct.lib”) /*link library*/
#   endif
# endif
#else
# define DECL_CSTR
#endif
#endif
```

2. Add a MODULE command to the config file in the library directory.

   This tells the build tool the module name and include path for the dcl_<*module*>.h file. It also tells the build tool that the library can be built as a shared/DLL library.

```
MODULE CSTR constrct/dcl_cstr.h
```

3. Add a DEFINE * EXPORT_<*MODULE*> command to the config file in the library directory.

   This causes the dcl_<*module*>.h file to define DECL_<*MODULE*> so that symbols are exported if ACIS_DLL is defined.

   ```
   DEFINE * EXPORT_CSTR
   ```

4. Add a LIB_DEPEND command to the config file in the library directory.

   This tells the build tool which libraries this library depends on, so it builds them in the correct order. The first library listed is the one being built, followed by those on which it depends.

   ```
   LIB_DEPEND constrct intersct kernel spline
   ```

5. Add the DECL_<*MODULE*> modifier to declarations of classes, global variables and functions in header files.

   This tells the compiler that the symbol should be exported when compiling files in the DLL and imported when compiling files outside the DLL.

   ```
   DECL_CSTR outcome api_initialize_constructors();
   class DECL_CSTR splgrid {
   .
   .
   .
   };
   ```

6. Change definitions of THIS_LIB and PARENT_LIB to <*MODULE*> instead of NONE.

   ```
   #define THIS() ATTRIB_INT
   #define THIS_LIB INTR
   #define PARENT() ATTRIB_BLND
   #define PARENT_LIB KERN
   ```

7. Change uses of the MODULE_DEF, MODULE_REF, ENTITY_FUNCTIONS, ATTRIB_FUNCTIONS, MASTER_ENTITY_DECL, MASTER_ATTRIB_DECL, DISPATCH_DECL, and LIST macros to specify <*MODULE*> instead of NONE.

   ```
   #define MODULE() sg_check_wire
   MODULE_DEF(”sg_check_wire”, CSTR);
   ```

8. Include the dcl_<*module*>.h file in any file which uses DECL_<*MODULE*> or <*MODULE*> as described above.

```
#include ”constrct/dcl_cstr.h”
```

9. Ensure that the header file that declares a symbol is included in the source file in which it is defined. Otherwise, you will get unresolved symbols when attempting to reference the symbol from outside the DLL.

These steps only make it possible to build the library as a shared/DLL library. The symbol ACIS_DLL must be defined during compilation and special commands must be used to create the DLL or shared library instead of a static library. These items should already be handled by the architecture specific config files in the bldcfg directory.

Note *These steps are not* required *for UNIX platforms. Steps 2 and 4 tell the build tool to build as a shared library, if allowed, and are the only steps needed for shared libraries on UNIX platforms.*

C Runtime Library DLL

Topic: *Building Applications, *SAT Save and Restore

When using the ACIS DLLs, it is important that you link your application against the DLL version of the C Runtime Library. Otherwise you will have two separate versions of the runtime library. Files opened (using fopen) by the C runtime library in your executable will not be recognized by the C runtime library used by ACIS and other DLLs. If you use the DLL version of the C Runtime Library, it will be shared by the executable and all DLLs.

You must also make sure that you do not use two different C runtime DLLs (e.g., one release and one debug) when using ACIS. When two different runtime DLLs are in use, several problems can occur. One problem is that file pointers can not be shared between two runtime DLLs. Each C runtime DLL has its own collection of file pointers (FILE*), and one C runtime DLL will experience an access violation if it tries to work with another’s file pointer. Another problem is that memory allocated by one C runtime DLL can not be deleted by another C runtime DLL, since each has its own memory heap.

A common case in which two different C runtime DLLs are used is the mixing of the debug and release versions of the C runtime DLLs. For example, if your application is built with the debug version of the C Runtime Library DLL (msvcrtd.dll), and your ACIS DLLs use the release DLL version (msvcrt.dll), then you will experience problems. This is a frequent source of access violations for save and restore.

You can determine which C runtime DLL is used by any DLL or executable file using the dumpbin program, which is part of the Visual C++ distribution. For example, to check your application executable, use (substitute the correct name of your application):

```
dumpbin /imports application.exe | findstr /i dll
```

To check your ACIS binaries, use (substitute the correct name/version of your Kernel DLL):

```
dumpbin /imports kern60.dll | findstr /i dll
```

If the result shows msvcrt.dll, that means the release DLL is used; if it shows msvcrtd.dll, that means the debug DLL is used (the added “d” means “debug version”). If one is using the release DLL and the other is using the debug DLL, you need to change either your application or your ACIS binaries so that they use the same C runtime DLL.