Chapter 9.
Entity Derivation Macros

Topic: *Entity, *Extending ACIS, *C++ Interface, *Attributes

The entity is a fundamental concept in ACIS, and the ENTITY class is the fundamental data structure in ACIS that supports roll back, copy, save/restore and error recovery. All persistent data types in an ACIS model are derived from ENTITY or are directly controlled by an ENTITY derived class. Derived ENTITYs make up the foundation of ACIS. Derivation of new ENTITY classes is also a common means of extending ACIS, and application developers can derive classes from ENTITY to define their own persistent data types. Attributes are special entities containing user-defined data that can be attached to other entities. ATTRIB is a derived class of ENTITY and is the base class for implementing attributes.

In exchange for the power of classes derived from ENTITY, the application programmer bears the burden of properly implementing a large number of functions (virtual functions of ENTITY). Fortunately ACIS provides a number of macros (defined in entity.hxx), that ease this burden considerably. There are macros to fully or partially implement each of the required virtual functions. Macros specific to attributes are defined in at_macro.hxx.

The use of these macros also protects the programmer from possible changes to the function signatures in different versions of ACIS. For example, in ACIS Release 5.0, a new argument was added to the copy_scan function. Only code that actually uses the type information needed to be modified. The majority of derived ENTITYs could simply be recompiled with the revised header file.

This chapter describes the macros used when deriving classes from ENTITY (or ATTRIB), the functions generated by the macros, when those functions are called, and how the developer should implement the code specific to his own ENTITY derivations. Attribute derivation is discussed in more detail in the attributes overview chapter of the *Kernel Component Manual*.

Macros You Must Define

Topic: *Entity, *Extending ACIS

For each class you derive from ENTITY, you must define (using #define) several simple macros. These define the name for the derived ENTITY and specify the class from which the new class is derived. The values are used by the other ENTITY derivation macros to build names of functions and other pieces of code. These macros are defined in the implementation file and must not interfere with other ENTITY derivations. If more than one ENTITY is implemented in the same file, you must undefine (#undef) the macros and redefine them.

These macros are:

THIS() . . . . . The name of this class being defined

PARENT() . . . . . The name of the immediate base class (parent)

THIS_LIB . . . . . The name of the module in which this class is implemented

PARENT_LIB . . . . . The name of the module in which the immediate base class (parent) is implemented

<*class*>_NAME . . . . . The entity's external identifier (<*class*> is the class name)

You must define THIS() and PARENT() with empty parentheses and THIS_LIB and PARENT_LIB without parentheses. A string must be provided for <class>_NAME.

Organization Entity Class

Topic: *Entity, *Extending ACIS

The "organization entity class" organizes the ENTITY derivation hierarchy and provides a name space (using a unique sentinel) for each customer or project. It is derived from the ENTITY class. The developer should derive application entities from his own organization entity class (also known as the *master entity*) and not directly from the ENTITY class. The sentinel name for the class will appear in the SAT file and assures that there is no confusion about how to restore the derived entity. Refer to Chapter 8, *Extending the Modeler*, for more information about sentinels and organization classes.

Note *Contact Spatial's customer support department to have a unique sentinel assigned to your ACIS development project, to insure that the chosen sentinel is not being used by any other customer or by ACIS.*

The following macros are used for constructing an application's master entity. This is the sentinel-level organization entity class that is constructed entirely by the macros:

MASTER_ENTITY_DECL . . . . . Used to declare a master (organization) entity class. Arguments include the class name, the component name, and the macros ENTITY_FUNCTIONS and LOOKUP_FUNCTION.

MASTER_ENTITY_DEFN . . . . . Used to implement a master (organization) entity class. Arguments include the extension ID and the macros ENTITY_DEF, LOOKUP_DEF, SAVE_DEF, RESTORE_DEF, COPY_DEF, SCAN_DEF, FIX_POINTER_DEF, and TERMINATE_DEF.

The following macros are used for constructing an application's master attribute. This is the sentinel-level organization attribute class that is constructed entirely by the macros:

MASTER_ATTRIB_DECL . . Declares an master (organization) attribute class, which has no data or methods of its own. This macro includes the macros defined by the ATTRIB_FUNCTIONS macro.

MASTER_ATTRIB_DEFN . . Generates all the code necessary to define an master (organization) attribute class.

Example Master Entity

This section contains code fragments for defining an example master entity, called ENTITY_XYZ (using sentinel "XYZ") from which other classes can then be derived. Both the header file and implementation file are shown. This example assumes the class is defined in a module (directory) named xyzmod. In section *Implementation Macros*, some short code fragments are used to illustrate the macros. These use an example entity, MY_ENTITY, which could be derived from the organization entity ENTITY_XYZ.

Header File

This example header file is named xyz.hxx.

C++ Example

```
#include ”kernel/acis.hxx”
#include ”kernel/kerndata/data/entity.hxx”
#include ”kernel/dcl_xyzmod.h”

#define ENTITY_XYZ_LEVEL ( ENTITY_LEVEL + 1)

// Identifier for the type of ENTITY
extern DECL_XYZMOD int ENTITY_XYZ_TYPE;

// Use ACIS macro to declare master entity
MASTER_ENTITY_DECL(ENTITY_XYZ, XYZMOD )
```

Example 9-1. xyz.hxx

Implementation File

This example implementation file is named xyz.cxx.

C++ Example

```
#include ”kernel/acis.hxx”

#include ”xyzmod/xyz.hxx”
#include ”kernel/kerndata/data/datamsc.hxx”

// This ENTITY is:
#define THIS() ENTITY_XYZ
#define THIS_LIB XYZMOD

// Base class is:
#define PARENT() ENTITY
#define PARENT_LIB KERN

// Name of entity:
#define ENTITY_XYZ_NAME ”xyz”

// Use ACIS macro to define master entity
MASTER_ENTITY_DEFN(”xyz master entity” )
```

Example 9-2. xyz.cxx

Implementation Macros

Topic: *Entity, *Extending ACIS

Macros that aid the implementation of some common functions (methods) for classes derived from ENTITY are defined in the ENTITY class header file entity.hxx. Macros for classes derived from ATTRIB are defined in at_macro.hxx

Generally, the *common* methods that do the main work are protected, as they are only called from the main entry functions or called by themselves higher up the derivation tree. The restore_common method is public, though, because the main restore function is not a method (member function), and making it a friend would prevent its being declared static (i.e., local to the defining module). The same utility routines defined for ENTITY may be overloaded for the derived class. Several of these routines require class-specific code, so several additional macros are declared to bracket that code; much of the required functionality is defined within macros.

This sequence of macros must appear in the entity implementation file before any other implementation routines—in particular before constructors and the object copy routine—but after the definition of THIS and PARENT.

This section describes the implementation macros:

ENTITY_DEF	UTILITY_DEF	DEBUG_DEF
SAVE_DEF	RESTORE_DEF	COPY_DEF
SCAN_DEF	FIX_POINTER_DEF	TRANSFORM_DEF
LOSE_DEF	DTOR_DEF	TERMINATE_DEF
SIMPLE_COPY_LOSE		

For each macro, the *signature* of the actual function being implemented is shown, where applicable, along with a discussion of when the function is called by the system and the proper technique for implementing it.

The macros required for any derivation from ENTITY are ENTITY_DEF, SAVE_DEF, RESTORE_DEF, COPY_DEF, SCAN_DEF, FIX_POINTER_DEF, and TERMINATE_DEF. The macro ATTCOPY_DEF may be used instead of ENTITY_DEF for any class derived from ATTRIB.

With the exception of ENTITY_DEF, ATTCOPY_DEF, and TERMINATE_DEF, all of the *_DEF macros supply a closing curly brace to terminate the definition of the previous function, and then supply the declaration portion of their own function. ENTITY_DEF and ATTCOPY_DEF are the starting macros (depending on whether this is for an ENTITY or ATTRIB), and should be the first macro used in any ENTITY or ATTRIB implementation. Thus, they do not supply a closing brace, because there is not a previous macro. TERMINATE_DEF simply provides the closing brace for the last macro.

ENTITY_DEF

Topic: *Entity, *Extending ACIS

ENTITY_DEF must be the first macro invoked in the implementation of a derived ENTITY. It invokes UTILITY_DEF and DEBUG_DEF. DEBUG_DEF is invoked last. Immediately following ENTITY_DEF, one should write code to implement debug_ent as described in the section for the DEBUG_DEF macro.

ATTCOPY_DEF

Topic: *Entity, *Extending ACIS

ATTCOPY_DEF is an alternative to the ENTITY_DEF macro that should be the first macro invoked for a derived ATTRIB. It invokes UTILITY_DEF and then starts the definition of fixup_copy. This macro can be used for regular ENTITYs if a fixup_copy function is needed.

UTILITY_DEF

Topic: *Entity, *Extending ACIS

UTILITY_DEF is invoked by ENTITY_DEF or by ATTCOPY_DEF. The programmer does not need to invoke it directly. It provides stock definitions for several required ENTITY functions and static data for the restore function and dynamic virtual functions.

DEBUG_DEF

Topic: *Entity, *Extending ACIS

The DEBUG_DEF macro calls the debug_ent function, which is used to produce neatly formatted, human readable information about the contents of the entity in the debug file. It is also used when gathering size data only, in which case the passed in file pointer will be NULL.

Function debug_ent is called from the debug_entity function, which prints out details of an entity and all of its subsidiary and sibling entities and from debug_lists, which prints subsidiary entities only. It is called from the entity:debug Scheme extension. It is also called from various points in the code that are inside DEBUG_LEVEL blocks.

Signature

```
void MY_ENTITY::debug_ent( FILE *fp ) const
```

Implementation

DEBUG_DEF should be implemented with functions declared at the end of file kernel/kerndata/data/debug.hxx. These functions are all named with the string “debug_”at their beginning of their name and the type of data at the end. For example, use debug_box to print a formatted description of SPAbox data. Functions are provided for most data types commonly used in ENTITY derivations. For other types, use debug_title followed by a call to fprintf guarded by a check that the file pointer is not NULL.

Three functions are provided for formatting entity pointers. Select the one to use based on the notion that when debugging a substructure, one would like to see its subsidiary entities, but not the containing entities. For example, when debugging a face, one would like to look down the structure to the loops, but not up the structure to the shell, from which many other faces could be dragged in. The difference between the three functions is whether the entity will be added to the list of entities to be debugged later in the same call to debug_entity. The three functions are:

debug_new_pointer . . . . . Used for subsidiary entities; always adds to the list so that subsidiary entities will be included in the debug output.

debug_old_pointer . . . . . Used for containing entities; never adds to the list, so that containing entities are not included.

debug_sib_pointer . . . . . Used for entities at the same level, such as the next_in_list pointer from a face; adds to the list when the global flag, debug_siblings is true. This flag is controlled by whether one starts with debug_entity or debug_lists.

A problem exists with attributes that point to entities in another body: following those pointers would cause entities of the other body to be mixed in with those of the entity of interest, causing great confusion. Use debug_old_pointer to write pointers into other bodies or data structures.

SAVE_DEF

Topic: *Entity, *Extending ACIS

The save_common method is called when saving to storage by any API whose name begins with “api_save_entity_list”.

Signature

```
void MY_ENTITY::save_common(ENTITY_LIST& list) const
```

Implementation

Implement SAVE_DEF with functions that are declared in kernel/kernutil/fileio/fileio.hxx and have names beginning with “write_”. For entity pointers, use write_ptr, which is declared in kernel/kerndata/savres/savres.hxx.

Functions are provided for all the basic types and for some compound types, such as SPAvector and SPAtransf. Other compound types, including user defined structures, should be saved using several calls to save the more basic types. Functions write_logical and write_enum allow the user to specify strings to be saved instead of simple numeric values. This makes the SAT file more human readable. The strings are ignored for binary saves. Function write_ptr is used to save entity pointers. The list is used to map the pointers to an index within the SAT file.

Versioning

ACIS allows an application to save in the format of any lower level ACIS version. The version being saved is stored in the global variable, save_version_number (this can be set with an option). File kernel/kerndata/savres/versions.hxx defines the version when the SAT format changes for any *Spatial* supplied entities.

RESTORE_DEF

Topic: *Entity, *Extending ACIS

The restore_common method is called when reading from storage by any API whose name begins with “api_restore_entity_list”.

Signature

```
void MY_ENTITY::restore_common()
```

Implementation

Implement RESTORE_DEF with functions declared in kernel/kernutil/fileio/fileio.hxx with names beginning with “read_”. For entity pointers, use read_ptr, which is declared in kernel/kerndata/savres/savres.hxx.

Functions are provided for all the basic types and for some compound types, such as SPAvector and SPAtransf. Other compound types, including user defined structures, should be restored using several calls to restore the more basic types. Functions read_logical and read_enum allow the user to specify strings to be restored instead of simple numeric values. The strings must match what was specified in the save function. The strings are ignored for binary restores. Function read_ptr is used to restore entity pointers. The list is used to map the pointers to an index within the SAT file.

Versioning

ACIS allows an application to restore data in the format of any lower level ACIS version. The version being restored is stored in the global variable, restore_version_number (this can be set with an option). File kernel/kerndata/savres/versions.hxx defines the version when the SAT format changes for any *Spatial* supplied entities.

COPY_DEF

Topic: *Entity, *Extending ACIS

COPY_DEF defines the copy_common method, which is called during execution of api_copy_entity to transfer information from the “from” entity to “this” entity. The list is used to map entity pointers to an index in the copy array.

Signature

```
void MY_ENTITY::copy_common( ENTITY_LIST &list, MY_ENTITY const *from )
```

Implementation

Most data can be copied by a simple assignment. For example:

```
my_data = from->my_data;
```

Pointers to ENTITYs are handled specially, as they need to be re-mapped to point to the copy of the original data. For example, to copy a pointer to MY_ENTITY use:

```
my_entity_ptr = (MY_ENTITY *)list.lookup( from->my_entity_ptr );
```

SCAN_DEF

Topic: *Entity, *Extending ACIS

SCAN_DEF defines the copy_scan method, which is used in many places in the system to gather a list of connected entities. APIs api_copy_entity and api_del_entity use the list to copy or delete all of the connected entities.

Signature

```
void MY_ENTITY::copy_scan( ENTITY_LIST &list, SCAN_TYPE reason ) const
```

Implementation

Most derived entities will simply add each entity pointer to the list. Introduced in ACIS Release 5.0 is the SCAN_TYPE reason code. This allows an application to have different behavior depending on the reason copy_scan is being called. The reason codes are defined in an enumerated type:

```
enum SCAN_TYPE {
    SCAN_UNSPECIFIED,
    SCAN_COPY,      // The default for backward compatibility
    SCAN_DELETE,
    SCAN_DISTRIBUTE,
    SCAN_DEEP_COPY,
    SCAN_PATTERN,
    SCAN_PATTERN_DOWN
};
```

FIX_POINTER_DEF

Topic: *Entity, *Extending ACIS

FIX_POINTER_DEF defines the fix_common method, which is called during the final stage of restore and copy to map indices into any array of entities to actual pointer values.

Signature

```
void MY_ENTITY::fix_common( ENTITY *array[] )
```

Implementation

When fix_common is called, entity pointers within the derived entity contain indices into an array of entities. These can be mapped to actual pointers with code like the following.

```
my_entity_ptr = (MY_ENTITY *)read_array( array, my_entity_ptr )
```

Versioning

ACIS allows an application to restore data in the format of any lower level ACIS version. The version being restored is stored in the global variable, restore_version_number (this can be set with an option). File kernel/kerndata/savres/versions.hxx defines the version when the SAT format changes for any *Spatial* supplied entities. During copy, the restore_version_number will be set to the most recent version of ACIS.

TRANSFORM_DEF

Topic: *Entity, *Extending ACIS

TRANSFORM_DEF defines apply_transform, which is an optional function that should apply the given transform to the entity.

Signature

```
logical MY_ENTITY::apply_transform( const SPAtransf &tform,
    ENTITY_LIST &list, logical negate )
```

Implementation

When overriding the apply_transform virtual function, one should use the TRANSFORM_FUNCTION macro in the ENTITY declaration.

The application programmer should provide code to apply the transform to the entity and call apply_transform on any subsidiary entities. The list argument is used in the macro code to prevent duplicate application of the transform to the same entity. The application writer normally does not need to use the list argument other than to pass it on to the subsidiary entities. The negate argument indicates that the body is being negated as well as transformed.

LOSE_DEF

Topic: *Entity, *Extending ACIS

LOSE_DEF defines the lose method, which is called to tell the system that the ENTITY is no longer part of the active model, but its data should be saved for use in a future roll back operation. It is the user-level destructor for an ENTITY. The actual destructor is declared as protected by UTILITY_DEF, so its use causes a compile time error.

Signature

```
void MY_ENTITY::lose()
```

Implementation

This must do whatever is required to lose dependent ENTITYs or otherwise cleanup. LOSE_DEF should always be followed by DTOR_DEF, which supplies a call to PARENT::lose as the last line of the preceding lose function.

The ENTITY could be restored via roll back, so one cannot yet delete non-ENTITY data that may be pointed to, unless it can be easily recalculated. For example, TEXT_ENTITY does not delete its string in the lose function because the information would be permanently lost. ENTITYs with boxes generally do delete the box, which can be recalculated on demand.

DTOR_DEF

Topic: *Entity, *Extending ACIS

DTOR_DEF defines the ENTITY's destructor, which is called to permanently release all memory back to the operating system. It is called by ENTITY's only friend, BULLETIN, when the BULLETIN is deleted and there is no longer any chance that the ENTITY could be restored via roll back

Signature

```
MY_ENTITY::~MY_ENTITY
```

Implementation

DTOR_DEF should always follow LOSE_DEF, as it supplies the final call to the parent ENTITY's lose method. The implementation must do whatever is required to release any non-ENTITY data pointed to by the ENTITY, but should not delete subsidiary ENTITYs (they will be deleted by separate calls to the BULLETIN destructor).

TERMINATE_DEF

Topic: *Entity, *Extending ACIS

TERMINATE_DEF supplies the final closing curly brace for the last *_DEF macro used.

SIMPLE_COPY_LOSE

Topic: *Entity, *Extending ACIS

The SIMPLE_COPY_LOSE macro supplies the lose and fixup_copy methods and the destructor for simple ENTITYs that do not need any special handling for updating connected objects or any dependent structures.

Implementation

Simply invoke the macro. No further implementation is required. SIMPLE_COPY_LOSE invokes the three macros SIMPLE_COPY, SIMPLE_LOSE, SIMPLE_DTOR.