Chapter 5. Functions

Topic: Ignore

The function interface is a set of Application Procedural Interface (API) and Direct Interface (DI) functions that an application can invoke to interact with ACIS. API functions, which combine modeler functionality with application support features such as argument error checking and roll back, are the main interface between applications and ACIS. The DI functions provide access to modeler functionality, but do not provide the additional application support features, and, unlike APIs, are not guaranteed to remain consistent from release to release.

This chapter describes the functions for the Scheme Support Component. It contains an alphabetical list of reference templates that describe each function. Refer to the *3D ACIS Online Help User's Guide* for a description of the fields in the reference template.

active_part_context

Function: Part Management

Action:	Gets the active PART_CONTEXT. Use this method when adding a new ENTITY to a PART.
Prototype:	`PART_CONTEXT* active_part_context ();`

Includes:	`#include ”kernel/acis.hxx”`
	`#include ”pmhusk/part_ctx.hxx”`
	`#include ”scmapp/scmapp.hxx”`

Description:	Refer to Action.
Errors:	None
Limitations:	None
Library:	scmapp
Filename:	scm/scmapp/scmapp.hxx
Effect:	Read–only

api_pm_add_entity

Function: Part Management

Action: Adds an ENTITY to a PART.

Prototype:
```
outcome api_pm_add_entity (
    ENTITY* entity,          // entity to be added
    PART* part               // part to which to add
                             // entity
    );
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/data/entity.hxx”
#include ”part/pmhusk/part.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API adds a specified entity to a specified part. If the entity is already in a different PART, it is first removed from the old part. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/pmhusk/api/pm_api.hxx

Effect: Changes model

api_pm_create_part

Function: Part Management

Action: Creates a new PART.

Prototype:
```
outcome api_pm_create_part (
    unsigned int,            // initial size of entity
                             // table for part
    PART*& part              // returns part
    );
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”part/pmhusk/part.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description:	This API creates a new part. It initially allocates enough space to contain the specified size (the number of entities). All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.
Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/api/pm_api.hxx
Effect:	Changes model

api_pm_delete_all_states

Function: History and Roll, Part Management

Action: Deletes all states.

Prototype:

```
outcome api_pm_delete_all_states (
    HISTORY_STREAM* hs       // history stream to
        = NULL               // delete
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/bulletin/bulletin.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description:	This API deletes all operations defined using api_pm_start_state and api_pm_note_state for the given history stream. Use this API when clearing a part in preparation for loading or creating a new part. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.
Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/api/pm_api.hxx
Effect:	System routine

api_pm_delete_part

Function: Part Management

Action: Deletes a PART.

Prototype:

```
outcome api_pm_delete_part (
    PART* part              // part
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”part/pmhusk/part.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API deletes the specified part. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/pmhusk/api/pm_api.hxx

Effect: Changes model

api_pm_entity_id

Function: Part Management

Action: Gets the entity ID and part for an ENTITY.

Prototype:

```
outcome api_pm_entity_id (
    ENTITY* ent,            // entity to identify
    entity_id_t& id,        // entity to identify
    PART*& part             // part containing the
                            // entity
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/data/entity.hxx”
#include ”part/pmhusk/entityid.hxx”
#include ”part/pmhusk/part.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API returns the entity ID (id) and the part containing the specified entity. If the entity is not in the part, this API returns the entity ID (id) as 0 and the part as NULL. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/api/pm_api.hxx
Effect:	Read–only

api_pm_load_part

Function: Part Management

Action: Loads a file into a PART.

Prototype:

```
outcome api_pm_load_part (
    FILE* fp,                     // file containing
                                  // entities to load
    logical text_mode,            // TRUE (text) or
                                  // FALSE (binary)
    PART* the_part,               // part in which to
                                  // load entities
    logical with_history,         // TRUE to restore
                                  // history if it
                                  // exists in the file
    ENTITY_LIST& new_entities     // returns list of
                                  // entities loaded
                                  // into part
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/lists/lists.hxx”
#include ”part/pmhusk/part.hxx”
#include ”scmext/load_part.hxx”
#include ”baseutil/logical.h”
```

Description: This API loads the entities defined in an open file fp into the specified part. The file must be open and positioned to the start of the entity data to be read. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/scmext/load_part.hxx

Effect: Changes model

api_pm_lookup_entity

Function: Part Management, Entity

Action: Gets an entity given an ID and a PART.

Prototype:
```
outcome api_pm_lookup_entity (
    entity_id_t id,          // entity ID
    PART* part,              // part in which to look
                             // for entity
    ENTITY*& ent             // found entity
    );
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/data/entity.hxx”
#include ”part/pmhusk/entityid.hxx”
#include ”part/pmhusk/part.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API looks up an entity in a part given an entity id. If id does not exist in the part, this API returns NULL. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/pmhusk/api/pm_api.hxx

Effect: Read–only

api_pm_name_state

Function: History and Roll, Part Management

Action: Names the current state.

Prototype:
```
outcome api_pm_name_state (
    const char* name,        // name to give to
                             // current operation
    HISTORY_STREAM* hs       // returns history stream
        = NULL
    );
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/bulletin/bulletin.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API assigns a name to the recent operation. Call api_pm_name_state immediately after api_pm_note_state and before opening the next sate with to api_pm_start_state. api_pm_name_state names the most recent noted state. api_pm_name_state can also be called immediately following starting the modeler if it were desired that the “root” state be named. Use the specified name in calls to api_pm_roll_to_state to roll to the start of the current operation. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/pmhusk/api/pm_api.hxx

Effect: System routine

api_pm_note_state

Function: History and Roll, Part Management

Action: Marks the end of a state.

Prototype:
```
outcome api_pm_note_state (
    outcome out,            // outcome of operation
    int& depth              // depth of operation
                            // nesting after call
    );
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API marks the end of an operation. Match calls to api_pm_note_state with earlier calls to api_pm_start_state. Pairs can be nested to create larger operations. A new delta state is created for the outermost call only. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

The calls to api_pm_start_state and api_pm_note_state must be strictly paired regardless of errors. Start state and note state are paired by the use of a static level counter. If the note state were skipped when there was an error, the counter would be off by one and subsequent states would not be noted.

```
int depth;
api_pm_start_state(depth);
API_BEGIN

result = api_do_stuff_1(args);
check_outcome(result);     // If result is not ok,
                           // jump to API_END

// Alternate style of using check_outcome
check_outcome(api_do_stuff_2(args));

// Tell the part manager and graphics what happened
record_entity(new top level entity);
update_entity(modified top level entity);

API_END
api_pm_note_state(outcome(API_SUCCESS), depth);
```

If an error occurs, it will be caught by API_END. The api_pm_note_state is always called regardless of error. Note that the outcome is checked before recording or updating entities, so the part manager and graphics don't see anything bad.

It is also acceptable to use API_SYS_BEGIN/END or EXCEPTION_BEGIN/TRY/CATCH/END with api_pm_start_state in the EXCEPTION_BEGIN block and api_pm_note_state in an EXCEPTION_CATCH(TRUE) block.

Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/api/pm_api.hxx
Effect:	System routine

api_pm_part_entities

Function: Part Management

Action: Gets a list of entities in a PART.

Prototype:

```
outcome api_pm_part_entities (
    PART* part,              // part from which to get
                             // entities
    entity_filter* filter,   // filter used to select
                             // entities or NULL
    ENTITY_LIST& ent         // returns list of
                             // entities found
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/geomhusk/efilter.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/lists/lists.hxx”
#include ”part/pmhusk/part.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API returns the list of entities fount in a part that match the specified filter. If filter is NULL, this API returns all entities in the part. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/pmhusk/api/pm_api.hxx

Effect: Read–only

api_pm_remove_entity

Function: Part Management

Action: Removes an ENTITY from a part.

Prototype:

```
outcome api_pm_remove_entity (
    ENTITY* entity           // entity to be removed
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/data/entity.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API removes an ENTITY from a part. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/api/pm_api.hxx
Effect:	Changes model

api_pm_roll_n_states

Function: History and Roll, Part Management

Action: Rolls forward or backward a specified number of states.

Prototype:
```
outcome api_pm_roll_n_states (
    int n_wanted,             // number of states to
                              // roll
    HISTORY_STREAM* hs,       // history stream to roll
    int& n_actual             // returns actual number
                              // of states rolled
    );
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/bulletin/bulletin.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API rolls a specified number (n_wanted) of states. A negative number rolls to an earlier state; a positive number rolls to a later state. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/api/pm_api.hxx
Effect:	System routine

api_pm_roll_to_state

Function: History and Roll, Part Management

Action: Rolls to the start of a named state.

Prototype:
```
outcome api_pm_roll_to_state (
    const char* name,          // name of state to which
                               // to roll
    HISTORY_STREAM* hs,        // history stream
    int& n_actual              // number of states
                               // actually rolled
    );
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kerndata/bulletin/bulletin.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API rolls to the start of a named operation (name). If multiple operations have the same name, the latest one before the current state is used. If no operations with the given name occur before the current state, the first one after the current state is used. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/pmhusk/api/pm_api.hxx

Effect: System routine

api_pm_save_part

Function: Part Management

Action: Saves a PART to a file.

Prototype:
```
outcome api_pm_save_part (
    FILE* fp,                  // file in which to save
                               // entities
    logical text_mode,         // TRUE (text) or
                               // FALSE (binary)
    PART* the_part,            // PART containing
                               // entities to save
    logical with_history       // TRUE to save history
        = 0,                   // stream to the file
    logical mainline_only      // TRUE to ignore rolled
        = 0                    // states
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”part/pmhusk/part.hxx”
#include ”pmhusk/api/pm_api.hxx”
#include ”baseutil/logical.h”
```

Description: This API saves the entities contained in a PART to an open file (fp). The file must be open and positioned to the location to which the entities are to be written.

If the optional with_history is specified as TRUE, roll back history data will be saved as well. If the optional mainline_only flag is specified as TRUE, only un–rolled states will be saved to the file. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/pmhusk/api/pm_api.hxx

Effect: Changes model

api_pm_start_state

Function: History and Roll, Part Management

Action: Marks the start of a state.

Prototype:

```
outcome api_pm_start_state (
    int& depth              // depth of nesting of
                            // operations after call
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”pmhusk/api/pm_api.hxx”
```

Description: This API marks the start an operation. Match calls to api_pm_start_state with later calls to api_pm_note_state. Pairs may be nested to create larger operations. A new delta state is started for the outermost call only. All api_pm functions should be thought of as requiring the use of the PART_CONTEXT class.

The calls to api_pm_start_state and api_pm_note_state must be strictly paired regardless of errors. Start state and note state are paired by the use of a static level counter. If the note state were skipped when there was an error, the counter would be off by one and subsequent states would not be noted.

```
int depth;
api_pm_start_state(depth);
API_BEGIN

result = api_do_stuff_1(args);
check_outcome(result);     // If result is not ok,
                           // jump to API_END

// Alternate style of using check_outcome
check_outcome(api_do_stuff_2(args));

// Tell the part manager and graphics what happened
record_entity(new top level entity);
update_entity(modified top level entity);

API_END
api_pm_note_state(outcome(API_SUCCESS), depth);
```

If an error occurs, it will be caught by API_END. The api_pm_note_state is always called regardless of error. Note that the outcome is checked before recording or updating entities, so the part manager and graphics don't see anything bad.

It is also acceptable to use API_SYS_BEGIN/END or EXCEPTION_BEGIN/TRY/CATCH/END with api_pm_start_state in the EXCEPTION_BEGIN block and api_pm_note_state in an EXCEPTION_CATCH(TRUE) block.

Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/api/pm_api.hxx
Effect:	System routine

delete_GC_Objects

Function: Scheme Interface, Filtering

Action: Deletes the GC_Object.

Prototype:	`void delete_GC_Objects ();`

Includes:	`#include ”kernel/acis.hxx”`
	`#include ”scheme/gc_obj.hxx”`

Description:	Refer to Action.
Errors:	None
Limitations:	None
Library:	scheme
Filename:	scm/scheme/gc_obj.hxx
Effect:	System routine

get_part_context

Function: Part Management

Action: Gets the PART_CONTEXT from an ENTITY.

Prototype:

```
PART_CONTEXT* get_part_context (
    const ENTITY* ent       // given entity
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”kernel/kerndata/data/entity.hxx”
#include ”pmhusk/part_ctx.hxx”
```

Description:	Refer to Action.
Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/part_ctx.hxx
Effect:	Read–only

get_scheme_error_callback_list

Function: Scheme Interface, Callbacks

Action: Gets a global list of scheme error callbacks.

Prototype:
```
scheme_error_callback_list&
get_scheme_error_callback_list ();
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”scheme/err_cb.hxx”
```

Description: Refer to Action.

Errors: None

Limitations: None

Library: scheme

Filename: scm/scheme/err_cb.hxx

Effect: Read–only

get_Scm_String

Function: Scheme Interface, Text

Action: Creates a C++ const char* from a Scheme string object.

Prototype:
```
const char* get_Scm_String(
    ScmObject s               // Scheme object
    );
```

Includes:
```
#include ”kernel/acis.hxx”
#include ”scheme/elk/object.h”
#include ”scheme/scheme.hxx”
```

Description: This function always reuses the same space, so if it is necessary to retain the string for future use, copy it into another space.

Errors: None

Limitations: None

Library: scheme

Filename: scm/scheme/scheme.hxx

Effect: System routine

is_Scm_Real_List

Function: Scheme Interface, Mathematics

Action: Determines if a Scheme object is a list of reals.

Prototype:

```
logical is_Scm_Real_List (
    ScmObject list           // Scheme object
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”scheme/elk/object.h”
#include ”baseutil/logical.h”
#include ”scheme/scheme.hxx”
```

Description: Refer to Action.

Errors: None

Limitations: None

Library: scheme

Filename: scm/scheme/scheme.hxx

Effect: Read–only

refresh_all

Function: Viewing

Action: Refreshes all views.

Prototype:

```
void refresh_all();
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”pmhusk/part_ctx.hxx”
```

Description: Refreshes all views.

Errors: None

Limitations: None

Library: pmhusk

Filename: scm/pmhusk/part_ctx.hxx

Effect: System routine

SchemeEvaluate

Function: Scheme Interface

Action: Evaluates a string or Scheme object.

Prototype:

```
int SchemeEvaluate (
    ScmObject expr,          // command string
    ScmObject& result        // returns result
    );

int SchemeEvaluate (
    const char* str          // command string
    );

int SchemeEvaluate (
    const char* str,         // command string
    ScmObject& result        // returns result
    );

int SchemeEvaluate (
    const char* str,             // command string
    param_string& result_string // returns result
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”scheme/parm_str.hxx”
#include ”scheme/elk/object.h”
#include ”scheme/scm_eval.hxx”
```

Description: This function is overloaded.

First, it evaluates an expression given by a character string, and returns the result as a param_string. The param_string is cast to a char*. The function returns 0 if the procedure successfully evaluates; otherwise, it returns an exception code.

Second, it evaluates an expression given as a character string. This is useful for evaluating the expression for its side effects. The function does not return the result of evaluating the expression. The function returns 0 if the procedure successfully evaluates; otherwise, it returns an exception code.

Third, it evaluates a Scheme expression for its result, returned as a Scheme object. This version of SchemeEvaluate evaluates an expression that is given as a character string, and returns the result as a Scheme object. This has the benefit of not having to convert the result to a char* and back with the resulting potential for loss of precision. The is_Scm_<type> and get_Scm_<type> functions are called to check the result and convert it into a C++ object. The function returns 0 if the procedure successfully evaluates; otherwise, it returns an exception code.

Fourth, it evaluates a Scheme expression that is already a Scheme object. This version of SchemeEvaluate accepts a Scheme expression that has already been parsed into a Scheme object. Because it does not have to go through the Scheme reader to convert a string into an object, it is faster to evaluate the same expression many times. The expression to be evaluated must be protected from garbage collection if the repeated evaluation is not done within the scope of a single C++ procedure. The function returns 0 if the procedure successfully evaluates; otherwise, it returns an exception code.

Errors: None

Limitations: None

Library: scheme

Filename: scm/scheme/scm_eval.hxx

Effect: System routine

SchemeLoad

Function: Scheme Interface

Action: Loads a Scheme file into memory.

Prototype:

```
int SchemeLoad (
    const char* filename    // file to load
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”scheme/scm_eval.hxx”
```

Description: The function returns 0 if the loaded procedure successfully evaluates; otherwise, it returns an exception code.

Errors: None

Limitations: None

Library: scheme

Filename: scm/scheme/scm_eval.hxx

Effect: System routine

scheme_process

Function: Scheme Interface

Action: Builds and evaluates a Scheme command, optionally echoing the prompt and result.

Prototype:

```
int scheme_process (
    const char* inpLine,     // command string
    int echo                 // echo
    );
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”scheme/sprocess.h”
```

Description: scheme_process is called repeatedly with input lines that partially form a Scheme command. After each invocation, the function returns the current nesting level of parenthesis. When it has compiled a complete Scheme command with matching parentheses and quotes (the nesting level returns as 0), it evaluates the command by calling do_scheme. All Scheme procedures in the input string are evaluated before returning.

Errors: None

Limitations: None

Library: scheme

Filename: scm/scheme/sprocess.h

Effect: System routine

start_entity_creation

Function: Entity, History and Roll, Viewing

Action: Prepares for the definition of a new ENTITY.

Prototype:

```
void start_entity_creation ();
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”pmhusk/ent_utl.hxx”
```

Description: Use this routine in conjunction with start_entity_creation to bracket modifications to entities. start_entity_creation and end_entity_creation can be nested.

Using start_entity_creation, end_entity_creation, start_entity_modification, and end_entity_modification or any associated wrapper functions is not recommended. They sometimes cause more confusion than they are worth, because they hide some of what is going on. If code is written that operates on more than one entity, these don’t work. These will have to be broken out into their parts anyway.

Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/ent_utl.hxx
Effect:	System routine

start_entity_modification

Function: Entity, History and Roll, Viewing

Action: Prepares for ENTITY modification.

Prototype:

```
void start_entity_modification ();
```

Includes:

```
#include ”kernel/acis.hxx”
#include ”pmhusk/ent_utl.hxx”
```

Description: Use this routine in conjunction with end_entity_modification to bracket modifications to entities. start_entity_modification and end_entity_modification can be nested.

Using start_entity_creation, end_entity_creation, start_entity_modification, and end_entity_modification or any associated wrapper functions is not recommended. They sometimes cause more confusion than they are worth, because they hide some of what is going on. If code is written that operates on more than one entity, these don't work. These will have to be broken out into their parts anyway.

Errors:	None
Limitations:	None
Library:	pmhusk
Filename:	scm/pmhusk/ent_utl.hxx
Effect:	System routine