Chapter 6.
Error Handling and Messaging

Topic: *Error Handling

This chapter describes ACIS error handling and how to set up and use ACIS error messaging. Refer to Appendix A, *Error Messages*, for a list of error messages.

Error Handling

Topic: *Error Handling

Error handling transfers the control of the program from one place to another if unwanted results occur or a user interrupts the program. ACIS handles all standard errors that occur while executing a program. It also provides a procedure to add new error types. ACIS also provides a procedure that produces warnings at various stages of program execution.

Fatal errors are those from which there is no immediate recovery. Errors that are unexpected events, but are not immediately fatal, are *warnings*. The same event is a fatal error under some circumstances, unexpected but nonfatal under other circumstances, and acceptable under other circumstances. Regardless of whether a given event is fatal, the same error code is used. The function used with that error code informs the system whether the error was fatal.

Error Handling Functions and Macros

Topic: *Error Handling

The following error handling functions and macros are used by ACIS to catch and control errors. Applications may also need to use these when using the ACIS direct function or class interfaces (i.e., non-APIs). The functions and macros are listed in the order of execution:

error_begin . . . . . Establishes ACIS signal handling. Each call to error_begin must be offset by a corresponding call to error_end. Calls to these two routines may be nested. The outermost call to error_begin establishes signal handlers, resets the warning count, and resets the error hardness level. Each call to error_begin increments the error level.

error_harden . . . . . Inhibits processing of user interrupts. Each call to error_harden increments the error hardness level. User interrupts temporarily ignored while the error hardness level is greater than zero. All other signals and errors are processed normally.

error_soften . . . . . Enables processing of user interrupts. Each call to error_soften decrements the error hardness level. When the error hardness level reaches zero, any user interrupt that was ignored is processed.

error_end . . . . . Resets signal handling. Each call to error_end decrements the error level. When the error level reaches zero, error_end resets the signal handlers to those that were in effect when the corresponding call to error_begin was made. If a user interrupt was seen, The application's interrupt signal handler is called.

sys_error . . . . . Signals ACIS errors and interrupts. The errors reported using this function are fatal errors. If the crash option is on, the function causes a core dump. Otherwise, it transfers control to the innermost active ERROR_BEGIN macro, setting its error_no to the specified error code..

sys_warning . . . . . Reports unexpected, nonfatal events that occurred during the course of the execution. It saves the error code given as input in an err_mess_type global array. Like sys_error, this routine checks for the crash option. If the option is on, it prints the message associated with the current error code. Otherwise, the application must process this warnings.

Exception Handling Macros

Topic: *Error Handling

Four macros have been defined to allow ACIS code to take corrective action if an error or interrupt occurs. Applications may also need to use these when using the ACIS direct function or class interfaces (i.e., non-APIs). The four exception handling macros are:

```
EXCEPTION_BEGIN
    // Declarations of local variables to be cleaned up go here
EXCEPTION_TRY
    // Normal processing code goes here
EXCEPTION_CATCH(always_clean)
    // Interrupt/error cleanup code goes here
EXCEPTION_CATCH_FALSE
    // logically the same as EXCEPTION_CATCH(FALSE)
    // but it does not cause a warning about using a
    // constant expression in an if statement.
EXCEPTION_CATCH_TRUE
    // logically the same as EXCEPTION_CATCH(TRUE)
    // but it does not cause a warning about using a
    // constant expression in an if statement.
EXCEPTION_END_NO_RESIGNAL
    // logically the same as resignal_no = 0;
EXCEPTION_END
```

The macros must appear in the order specified above (BEGIN, TRY, CATCH, END). The EXCEPTION_CATCH macro is optional and may be omitted if only local variables are to be cleaned up. Sets of macros may be nested, but they must not overlap. The inner set must be fully contained between adjacent macros (normally between EXCEPTION_TRY and EXCEPTION_CATCH) of the outer set.

You must select zero or one of the CATCH macros and exactly one of the END macros in each exception block. For example, you should never have an EXCEPTION_CATCH_TRUE and an EXCEPTION_CATCH_FALSE in the same block of protected code.

The macros are defined in errorsys.hxx which is included in all .err files generated from .msg files. If a file currently calls sys_error, no new header files should be needed. Otherwise, errorsys.hxx should be included.

The EXCEPTION_BEGIN block (between the EXCEPTION_BEGIN and EXCEPTION_TRY macros) is used to declare variables that must be cleaned up if an error occurs. This includes pointers to be deleted in the EXCEPTION_CATCH block as well as local instances of classes that may contain pointers to potentially large amounts of dynamically allocated memory. Local instances are automatically destroyed by block exit code in the EXCEPTION_END macro before any error is re-signalled.

Variables declared in the EXCEPTION_BEGIN block should be declared as being “volatile” so that they will *not* be created as register variables. This will prevent the unexpected resetting of the variables during exception handling. Similarly, variables that are declared *before* the EXCEPTION_BEGIN block—and whose values are changed in the EXCEPTION_TRY block and are cleaned up in the EXCEPTION_CATCH block—are candidates to be made volatile. For example:

```
EXCEPTION_BEGIN
myclass* volatile vm = NULL;
EXCEPTION_TRY
vm = new myclass(...);
...
EXCEPTION_CATCH(TRUE)
delete vm;
EXCEPTION_END
```

If multiple variables are declared in the EXCEPTION_BEGIN block, their initialization should be kept as simple as possible since any error or interrupt that occurs in this block will not cause the corresponding EXCEPTION_CATCH block to be executed nor the local variables to be destroyed.

The EXCEPTION_TRY block (between the EXCEPTION_TRY and EXCEPTION_CATCH macros) contains the normal processing code. Variables declared here are visible only within the EXCEPTION_TRY block. They are destroyed by the block exit code only if no error or interrupt occurs. Variables that need to be cleaned up if an error occurs should be declared in or before the EXCEPTION_BEGIN block. Variables that must be visible after the EXCEPTION_END macro must be declared before the EXCEPTION_BEGIN macro.

The EXCEPTION_CATCH block (between the EXCEPTION_CATCH and EXCEPTION_END macros) is used to free dynamically allocated memory and reset global variables. The always_clean argument to the EXCEPTION_CATCH macro is a logical expression used to indicate whether the EXCEPTION_CATCH block should be executed even if no error occurs. This is useful to avoid duplication of code used to free temporary memory. The variable error_no can be examined to determine what (if any) error occurred. The variable resignal_no can be modified to change the error re-signalled to higher blocks. Setting resignal_no to zero stops the error from being re-signalled.

On most platforms, the use of setjmp and longjmp (for example, in macros API_BEGIN and API_END) has been replaced with the C++ try/catch statements. Some platforms do not support the try/catch statements, so ACIS uses setjmp and longjmp on those platforms (ACIS uses setjmp and longjmp if UNIX_EXCEPTION_TRAP is defined, and uses the try/catch statements if CPLUSPLUS_EXCEPTION_TRAP is defined).

Macro Example

Topic: *Error Handling

Without exception handling:

```
ENTITY_LIST list;              // Local instance with dynamic
                               // memory
...
EDGE *ent = new EDGE(...);     // ENTITY handled by BB mechanism
list.add(ent);
...
double *dbls = new double[n];  // Temporary memory
...
delete [] dbls;
```

With exception handling:

```
EXCEPTION_BEGIN
ENTITY_LIST list;
double *dbls = NULL;
EXCEPTION_TRY
...
EDGE *ent = new EDGE(...);
list.add(ent);
...
dbls = new double[n];
...
EXCEPTION_CATCH(TRUE)
delete [] dbls;
EXCEPTION_END
```

A parallel set of C_EXCEPTION_. . . macros is defined in except.h for use in C code. Due to the lack of constructors/destructors, local variables cannot be automatically cleaned up. Also, any statement (return, break, continue, goto) that would transfer control outside the EXCEPTION_BEGIN/EXCEPTION_END block should not be used.

Guidelines

Topic: *Error Handling

Following are some guidelines for implementing exception handling for memory cleanup These guidelines cover some of the most common situations involved in memory cleanup. There are bound to be other situations that are not covered here. These are only guidelines and individual circumstances may be better handled in other ways.

- Instances of classes derived from ENTITY and items pointed to by them do not need to be cleaned up. The bulletin board mechanism will take care of them. Cleaning up items pointed to by ENTITY class instances will almost certainly cause memory access errors.
- Simply looking for occurrences of new and delete will not identify all places where exception handling needs to be implemented. Functions that return pointers to allocated memory and instances of classes (such as ENTITY_LIST) that may contain pointers to large amounts of allocated memory should also be identified.

```
// Local instance of memory consuming class
ENTITY_LIST list;
// Function returns allocated memory
curve_curve_int *cci = int_cur_cur(c1, c2);
```

- Declarations of local instances of classes that may contain pointers to large amounts of dynamically allocated memory that would be freed by their destructors should be moved into a EXCEPTION_BEGIN block.
- All memory that the routine would normally free upon successful completion should be freed if an exception occurs.

```
FOO *foo_array = new FOO[n];
...
delete [] foo_array;
```

Becomes

```
EXCEPTION_BEGIN
FOO *foo_array = NULL;
EXCEPTION_TRY
foo_array = new FOO[n];
...
EXCEPTION_CATCH(TRUE)
delete [] foo_array;
EXCEPTION_END
```

- Memory that would be returned or attached to another object should be cleaned up in the EXCEPTION_CATCH block if it could exist for a significant period of time before being returned or attached. Avoid multiple deletion of allocated memory pointed to by class instances to be cleaned up.

```
FOO *foo_ptr = new FOO(...);
...
BAR *bar_ptr = new BAR(foo_ptr,...);
...
return bar_ptr;
```

Becomes

```
BAR *bar_ptr = NULL;
EXCEPTION_BEGIN
FOO *foo_ptr = NULL:
EXCEPTION_TRY
foo_ptr = new FOO(...);
...
bar_ptr = new BAR(foo_ptr,...);
...
EXCEPTION_CATCH(FALSE)
if (bar_ptr == NULL)
    delete foo_ptr;
else
    delete bar_ptr;
EXCEPTION_END
return bar_ptr;
```

- It is important to keep in mind that several type names (such as bs2_curve and bs3_surface) are actually pointers. Functions that return these types are probably returning pointers to allocated memory.
- Exception handling is relatively cheap, but not free. Consider size and duration of memory use when deciding whether to attempt to catch a potential leak.

```
// A small allocation, quickly attached
FOO *foo_ptr = new FOO(...);
bar.set_foo(foo_ptr);
```

User–Level Error Handling Functions

Topic: *Error Handling

In ACIS-based applications, precede the code that calls ACIS functions with API_BEGIN and succeed the program with API_END. These macros set up and terminate the error system automatically and are transparent to the application.

Error Printing Functions

Topic: *Error Handling

When an error is generated, an error code is returned as part of the outcome returned value, or as part of the warning list. The following functions are used to find or print error messages once an error has occurred:

find_err_entry	Uses the error code to return an error_table_entry. When an error is generated, an error code is returned as part of the outcome or as part of the warning list. The class error_table_entry contains the error code value, error code mnemonic, the corresponding error message, and the directory in which the error code was originally defined.
find_err_ident	Translates the error number to a string containing the mnemonic name associated with the given error number.
find_err_mess	Translates the error number to a string containing the message associated with the given error number.
find_err_module	Translates the error number to a string containing the name of the module associated with the given error number.
print_warnerr_mess	Prints the message associated with the current error number in a simple format for debugging purposes.
get_warnings	Obtains the warnings list.
init_warning	Resets the number of warnings to 0.

Error Return Mechanisms

Topic: *Error Handling

The outcome class contains a pointer to an error_info object. Each API has the option of returning additional error information in objects derived from error_info. Although no restriction is placed on the information these objects contain, new ENTITYs will be lost during roll back.

The base class error_info object contains class ID and object type methods, allowing the user to quickly determine the information available in a given error_info object. Each error_info object is allocated on the heap, and the outcome cleans up any error_info object it references.

Error System Process

1. At the start of each API, a global variable pointer to an error_info object is set to NULL.
2. Before sys_error is called, the global pointer is set to contain the relevant error_info object.
3. At the end of the API, before the outcome is returned, the global variable is examined, and if nonempty, the error_info is added to the outcome.

Two overloaded versions of the function sys_error set a global pointer to an error_info object. One version is passed an error_info object, and the other creates a standard_error_info object when sys_error is passed one or two ENTITYs. The standard_error_info class is derived from error_info, which provides error data that is adequate in a majority of cases, such as local operations and blending.

In the Local Ops, Remove Faces, and Shelling components, the error_info object returns an ENTITY that further specifies where the local operation first fails, when such information is available. A standard_error_info object is adequate for use in these components, and more detailed information could be returned, if necessary, by deriving a new class.

ACIS Error Messages

Topic: *Error Handling

ACIS provides a standard mechanism to create, link in, and use error codes and messages that must be used when adding error messages for ACIS. Tools are provided to automate the process under most platforms. Currently no tools exist for Macintosh platforms.

Adding New Error Messages

Topic: *Error Handling

Follow these steps to add new error messages:

1. Add the message to the <modname>.msg file for the module. Each line of this file consists of the mnemonic name of the message in capital letters, followed by the text of the message enclosed in quotes.

2. Run the build tool against the <modname>.msg file to create a <modname>.err and an e<modname>.cxx file. These are placed in a parallel directory called error.

3. Use the build tool to compile the e<modname>.cxx file.

4. Insert sys_error, sys_warning, and other macro and function calls into appropriate places in the code to handle error conditions.

5. Compile and link the executable using e<modname>.o.

Recording Error Messages

Topic: *Error Handling

Error messages for each module are recorded in a <modname>.msg file. The <modname>.msg file associates a mnemonic integer value (the error code) with a string (the error message). The following is a fragment of the api.msg file:

```
API_FAILED          ”operation unsuccessful”
EMPTY_ARRAY         ”array with no members given”
...

SMALL_RAD1          ”radius 1 is too small”
SMALL_RAD2          ”radius 2 is too small”
SMALL_LENGTH        ”length is too small”
...
```

When a new module is added, create a new <modname>.msg file.

Running the Error Message Tool

Topic: *Error Handling

After creating a message file, run the build tool. For example:

```
tools/bin/hp700/build api.msg
```

This tool reads the <modname>.msg file and creates a <modname>.err file containing a #define statement for each message. For example:

```
// Error code definitions for module ”kernapi/api”

#include ”kernutil/errorsys/errorsys.hxx”
#include ”error/message/errmsg.hxx”

extern message_module api_errmod;

#define API_FAILED api_errmod.message_code(0)
#define EMPTY_ARRAY api_errmod.message_code(1)
...

#define SMALL_RAD1 api_errmod.message_code(27)
#define SMALL_RAD2 api_errmod.message_code(28)
#define SMALL_LENGTH api_errmod.message_code(29)
...
```

The tool also creates an e<modname>.cxx file containing the definition of the error module for the messages in this module:

```
// Error code definitions for module ”kernapi/api”

#include ”acis.hxx”

#include <stdio.h>

#include ”kernutil/errorsys/errorsys.hxx”
#include ”error/message/errmsg.hxx”
```

```
static message_list api_msglst[] =
{
    {"API_FAILED", "operation unsuccessful"},
    {"EMPTY_ARRAY", "array with no members given"},
...

    {"SMALL_RAD1", "radius 1 is too small"},
    {"SMALL_RAD2", "radius 2 is too small"},
    {"SMALL_LENGTH", "length is too small"},
...

    {NULL, NULL}
};

message_module api_errmod("kernapi/api", api_msglst);
```

Compile this .cxx file using the build tool.

Macintosh Error Messages

Topic: *Error Handling

New error messages can be added by hand on Macintosh platforms. No tools currently support this process. Follow these steps to add new error messages using a Macintosh platform:

1. If this is a new module, create new error/<modname>.err and error/e<modname>.cxx files, using existing files as templates. Be sure to use a unique name for the message module declared at the end of the file.

2. Edit the error/<modname>.err file, adding a #define for each message to be added. This is easiest if all new messages are added at the end of the file, using the next number in sequence.

   ```
   #define MY_MESSAGE api_errmod.message_code(61)
   ```

3. Edit the error/e<modname>.cxx file, adding an entry to the message_list for each message to be added. The ordering of the entries in the message_list must match the numbers in the error/<modname>.err file and the entry with two NULL values must be the last entry in the list.

   ```
   {"MY_MESSAGE", "my error message"},
   ```

4. Compile the error/e<modname>.cxx file.