Chapter 1. Base Component

Component: *Memory Management, *Mathematics, *Error Handling, *Debugging

The Base Component (BASE), in the base directory, contains the code for low-level common functionality that is used by all ACIS components. BASE includes support for:

- Memory management
- Errors
- Exception Handling
- Messages
- Debug code
- Options
- Basic data types
- Multithreading

Although ACIS uses the base component, non-ACIS products can include just the Base Component instead of all of the kernel to reduce size. It can compile independently.

Memory Management

Topic: *Memory Management

The ACIS memory management functionality provides a mechanism to funnel all memory requests through a common pair of allocation and deallocation functions. These functions provide a common means for allocating and deallocating memory, collecting statistics about memory usage, auditing for memory leaks, and pattern filling.

The ACIS *memory manager* can also be used for managing application code memory, in addition to managing ACIS memory. The source code for the memory manager is shipped to all customers, allowing application developers to modify the memory management or statistics collection algorithms, or substitute commercial or proprietary memory management packages in their ACIS applications.

Note *The ability of the memory manager to capture all memory requests varies by platform according to the capabilities of individual compilers.*

ACIS memory management can:

- Classify memory by its persistence.
- Route memory requests to a common interface, consisting of two functions: acis_allocate and acis_discard.
- Allocate and deallocate actual memory, and allow this to be customizable by the application developer.
- Pattern fill unused memory with SNaN (Signal Not A Number) or other easily recognizable data to help spot references to uninitialized memory.
- Collect statistics about memory usage, such as who allocates memory, how much memory is being used, the maximum amount of memory used, and where memory leaks have occurred.
- Allow the application developer to implement memory management strategies using a proprietary or commercial memory manager.

Beginning with ACIS Release 5.3, ACIS memory management uses the capabilities of the interface to provide a fast allocation mechanism called free lists. By using interface data to determine optimum strategy, the memory manager can speed up allocation significantly. Instead of satisfying specific allocation requests from the system heap, the memory manager makes instance by instance decisions on use of the free list mechanism.

Initialization and Termination

Topic: *Memory Management

The memory manager is the first thing to be initialized when an application loads. It stays active until the last block of memory that was allocated has been freed, then terminates.

Configuration Choices

Topic: *Memory Management

Beginning with ACIS Release 7.0, the auditing and freelist capabilities of the ACIS memory management system have been delayed until run–time.

Changes to configuration are made through arguments to the initialize_base routine, as the following code snippet shows:

```
void *my_malloc( size_t size ) { return malloc(size); }
void my_free( void *ptr) { free( ptr); }

int main(int argc, char* argv[]) {
    logical use_freelists = FALSE;
    logical audit_leaks = TRUE;
initialize_base( my_malloc, my_free, use_freelists, audit_leaks );
api_start_modeller();
    mmgr_statistics * my_stats = mmgr_debug_stats();
    api_stop_modeller();
    terminate_base();
    return 0;
}
```

If initialize_base is not called (or if it is called with the default NULL argument), then the memory manager configuration defaults are set according to whether the baseutil library was built in debug or production mode. These defaults (listed below) have been chosen to provide the expected desired behavior; only users with special needs should need to explicitly change the settings.

The base_configuration class is defined in base\baseutil\base.hxx. It contains data members that allow users to specify allocation and deallocation functions, as well as flags to turn on/off audit and freelist functionality. The following table lists the flags, their production and debug default values, and their meanings.

Name	**Production library default**	**Debug library default**	**Meaning**
enable_freelists	TRUE	FALSE	No freelists if FALSE
enable_audit_leaks	FALSE	TRUE	Turns on MMGR memory leak checking if TRUE
enable_audit_logs	FALSE	FALSE	Turns on MMGR logging of all allocations, deletions, and ENTITY::lose() calls

Table 1-1. Configuration Defaults

Corresponding run–time flags have been added to the Scheme AIDE and Test Harness Commands Component. These flags are as follows:

–auditleaks	–noauditleaks
–freelists	–nofreelists
–auditlogs	

Customizing Low Level Memory Management

Topic: *Memory Management

The actual allocation and deallocation strategy of low level ACIS memory management can be replaced by the application developer with other commercial or custom memory management packages. The delivered product is optimized for best performance, but if required, the application developer can modify the memory manager functionality during the configuration period, from `main` to `start_modeller`. Any memory allocation (like calling printf) will automatically set the system to default. The application can use the delivered system, install a raw allocator/deallocator which will be used instead of the ACIS calls, or install a complex allocator/deallocator, which completely bypasses ACIS memory management and requires the application to control all memory.

When the ACIS memory manager is enabled, all calls to allocate memory are funnelled through the low level acis_allocate function, and all calls to deallocate memory are funnelled through the corresponding acis_discard function.

An application supplying runtime configuration arguments to the initialize_base function can choose between two types of control. The first modifies the behavior of the delivered system and is invoked with the following function:

```
DECL_BASE logical initialize_base(
    void*(*allocator)(size_t),
    void(*destructor)(void*),
    logical enable_freelists,
    logical enable_audit_logs,
    logical enable_audit_leaks);
```

The second completely replaces the delivered system by supplying functions which are called directly for all memory requests. This completely circumvents the delivered system and all of its capabilities (freelists, auditing, etc.). All available arguments are supplied to these functions to allow the application to create like functionality. The complex allocator and destructor are installed by supplying them to the following function:

```
DECL_BASE logical initialize_base(
void*(*allocator)(size_t,AcisMemType,AcisMemCall,const char*,int),
    void(*destructor)(void*,AcisMemCall,size_t) );

void *my_complex_allocate( size_t alloc_size, AcisMemType alloc_type,
AcisMemCall alloc_call, const char * alloc_file, int alloc_line ) {
    return malloc( alloc_size);
}

void my_complex_discard( void * alloc_ptr, AcisMemCall alloc_call, size_t
alloc_size) {
    free( alloc_ptr);
}
```

```
int main(int argc, char* argv[]) {
initialize_base( my_complex_allocate, my_complex_discard);
api_start_modeller();
    api_stop_modeller();
    terminate_base();
    return 0;
}
```

Supplying runtime configuration selections to the initialize_base function is optional. The mmgr will by default choose the appropriate settings for a release or debug build.

Managing Application Memory

Topic: *Memory Management

By default, ACIS memory management is used only within the ACIS code. However, the application can also use ACIS memory management within the application code.

When an application wants to use ACIS memory management for its own C run–time functions (malloc, calloc, realloc, strdup, and free), the developer can replace these function calls with the following macro calls.

- ACIS_MALLOC(alloc_size)
- ACIS_CALLOC(alloc_num,alloc_size)
- ACIS_REALLOC(memblock,alloc_size)
- ACIS_STRDUP(orgstring)
- ACIS_FREE(alloc_ptr)

When a class in the application is derived from the ACIS_OBJECT base class, it inherits ACIS_OBJECT::new and ACIS_OBJECT::delete and its memory allocation is handled by the memory manager.

When the application needs to route global new and delete calls through the ACIS memory manager, it can directly replace the new with any variation of the ACIS_NEW macro, and directly replace delete with any variation of the ACIS_DELETE macro. For example,

```
double* ptr = new double;
```

would become

```
double* ptr = ACIS_NEW double;
```

ACIS_NEWs allocates memory with a session persistence, ACIS_NEWd allocates memory with a document persistence, and ACIS_NEWt allocates memory with a temporary persistence.

Capturing Memory Requests

Topic: *Memory Management

Some compilers support overloaded new and delete operators, and can support routing all memory requests to the memory manager. Other compilers have more limited capabilities, and support routing only a subset of memory calls to the memory manager.

Grouping compilers by capability, the types of memory operations routed to the memory manager for each group are as follows.

Each successive group represents a more comprehensive level of memory management, and each group encompasses both itself and lower level groups. For example, memory management functionality in Group 3 also includes the functionality in Group 2 and in Group 1.

Group 1 . . . . . . C run–time memory functions malloc, calloc, realloc, strdup, and free are routed through the memory manager.

Group 2 . . . . . . Class level new and delete operators are routed through the memory manager, as well as Group 1 memory requests.
ACIS classes are derived from the base class ACIS_OBJECT.

The ACIS_OBJECT::new and ACIS_OBJECT::delete operators in turn call the acis_allocate and acis_discard functions.

For memory not allocated to free lists, the acis_allocate and acis_discard functions currently call malloc, but can be replaced by the application developer to call other memory management systems, or can be tailored to do such things as memory usage auditing, garbage collection, etc.

Group 1 C run–time memory functions are routed through the memory manager.

If this is selected, a rebuild is required before the program can run.

Group 3 . . . . . . Class-level array new and array delete requests are routed through the memory manager, as well as Group 2 and Group 1 memory requests.

Group 4 . . . . . . Global new and array new, decorated with tracking information, are routed through the memory manager, as well as Group 3, Group 2, and Group 1 memory requests. At this level of memory management, *all ACIS* memory requests are routed through the memory manager (100% coverage).

At Group 4 or above, the developer can turn on or off the run–time directives for MMGR_DEBUG, NO_FREE_LISTS, and MMGR_AUDIT_LEAKS.

Gathering Cumulative Memory Statistics

Topic: *Memory Management

The run–time memory usage statistics are automatically gathered by the delivered system with the auditing runtime configuration argument set to TRUE. The data is collected into the following structure and can be retrieved with a call to mmgr_debug_stats as the following code snippet shows.

```
struct mmgr_statistics {

    size_t high_bytes;      // the high water mark
    size_t alloc_bytes;     // total bytes allocated
    size_t alloc_calls;     // number of allocation calls
    size_t free_bytes;      // total bytes freed
    size_t free_calls;      // number of free calls
    size_t size_array[257]; // most frequent allocations sizes
    size_t double_deletes;      // non–audited address delete count
    size_t mismatched_callers;  // array allocation –
                                // non array delete for example
    size_t mismatched_sizes;    // allocated as a foo – deleted as a bar

    mutex_resource mmgr_statistics_mutex;
};
```

Application programmers can also set a breakpoint in mmgr_debug to intercept calls to the memory manager and look at these statistics at run time. They can also edit the mmgr_debug function to insert their own statistics handling code.

If the memory manager is built with the MMGR_AUDIT_LEAKS compiler directive defined, the gathered statistics are also dumped to the mmgr.log file upon program termination.

Gathering Memory Auditing (Leak) Information

Topic: *Memory Management

The memory manager can gather information on memory leaks.

If the memory manager is built with the MMGR_AUDIT_LEAKS compiler directive defined, the mmgr_debug function attempts to pair each allocation with a corresponding deallocation, and records all unpaired allocations and unpaired deallocations (memory leaks.)

The statistics gathered for each leak include:

alloc_num	a running count of the number of allocations made
alloc_size	the size in bytes of the allocation made
alloc_type	the type of allocation made

alloc_call	the allocation call made
alloc_line	the source code line number where the call was made, if ACIS has been compiled with the DEBUG compiler directive
alloc_file	the source file name from which the call was made, if ACIS has been compiled with the DEBUG compiler directive

Dumping to a file can be turned off using the mmgr_log option. A detailed log file called mmgr.log containing the statistics and leak information is optionally generated at application exit. The filename can be changed using the mmgr_file option. The debug build log will also contain file and line information.

Application programmers can also set a breakpoint in mmgr_debug to intercept calls to the memory manager and look at these statistics at run time, or can edit the mmgr_debug function to insert their own statistics handling code.

Enabling and Disabling Memory Manager Log

Topic: *Memory Management

Creation of Memory Manager log file (mmgr.log) could be disbaled by using a value of FALSE for the enable_audit_leaks and enable_audit_logs options inside the initialize_base function. Conversely the mmgr.log file could be enabled by using a value of TRUE for the above options. However, it should be noted that the call to the intitialize_base function should happen before starting the modeler as shown in the following code snippet.

Note *This is an unique case where an ACIS function is called before starting the modeler.*

```
base_configuration* base_config = new base_configuration();
base_config–>enable_audit_leaks = FALSE;
base_config–>enable_audit_logs = FALSE;
initialize_base(base_config);
delete base_config;
/*–––––––––––––––––––––––––––––––––––––––––––––––––––––*/
// INITIALIZE THE MODELER.
/*–––––––––––––––––––––––––––––––––––––––––––––––––––––*/
int memory_size = 0;   // size of model working space in bytes
outcome result;        // stores result of api calls
result = api_start_modeller(memory_size);
```

Pattern Filling Unused Memory

Topic: *Memory Management

If MMGR_DEBUG is defined, the memory manager pattern-fills unused memory blocks after discard with a strategic pattern. This pattern is defined in the mmgr.cxx file, which is shipped as readable source code to all customers. Applications can change this pattern by changing the following values and rebuilding the memory manager. The default pattern is an SNaN, which causes a segmentation violation signal to be generated if the program tries to reference the Not–a–Number pattern in unused memory.

```
LOCAL_CONST short ALLOC_PATTERN = 0x7ff1;
LOCAL_CONST short DEALLOC_PATTERN = 0x7ff7;
```

The ACIS option mmgr_fill can be used by the application to set a different fill pattern.

This is primarily a debugging tool. It can significantly slow the application.

Memory Persistence Categories

Topic: *Memory Management

Memory requests are classified into one of three *persistence* categories:

- *Temporary memory* implies that the block will be deallocated before the flow of execution leaves the scope of ACIS.
- *Document memory* is allocated when a document is opened and is deallocated when a document is closed. All data belonging to a body or bodies of a given document must be requested as Document memory and must not contain references to any other type of memory other than Document.
- *Session memory* has a lifetime equal to that of the application.

Free List Mechanism

Topic: *Memory Management

When a new block of memory is needed, it is allocated from the system and added to the appropriate free list. ACIS free list objects allocate 4KB blocks of memory, parceled into slots of 16, 32, 48, 64, 96, or 128 bytes respectively, as needed. Based on analysis of typical ACIS usage, these six sizes have been determined to be most effective.

For example, when the program requests 50 bytes of memory, the memory manager looks for a slot of the next largest block size, in this case 64 bytes. If an empty 64 byte slot does not currently exist, a 4KB block of memory is allocated from the operating system and added to the 64 byte-sized free list object. Then a pointer to the first free 64 byte slot within that block is returned to the requestor.

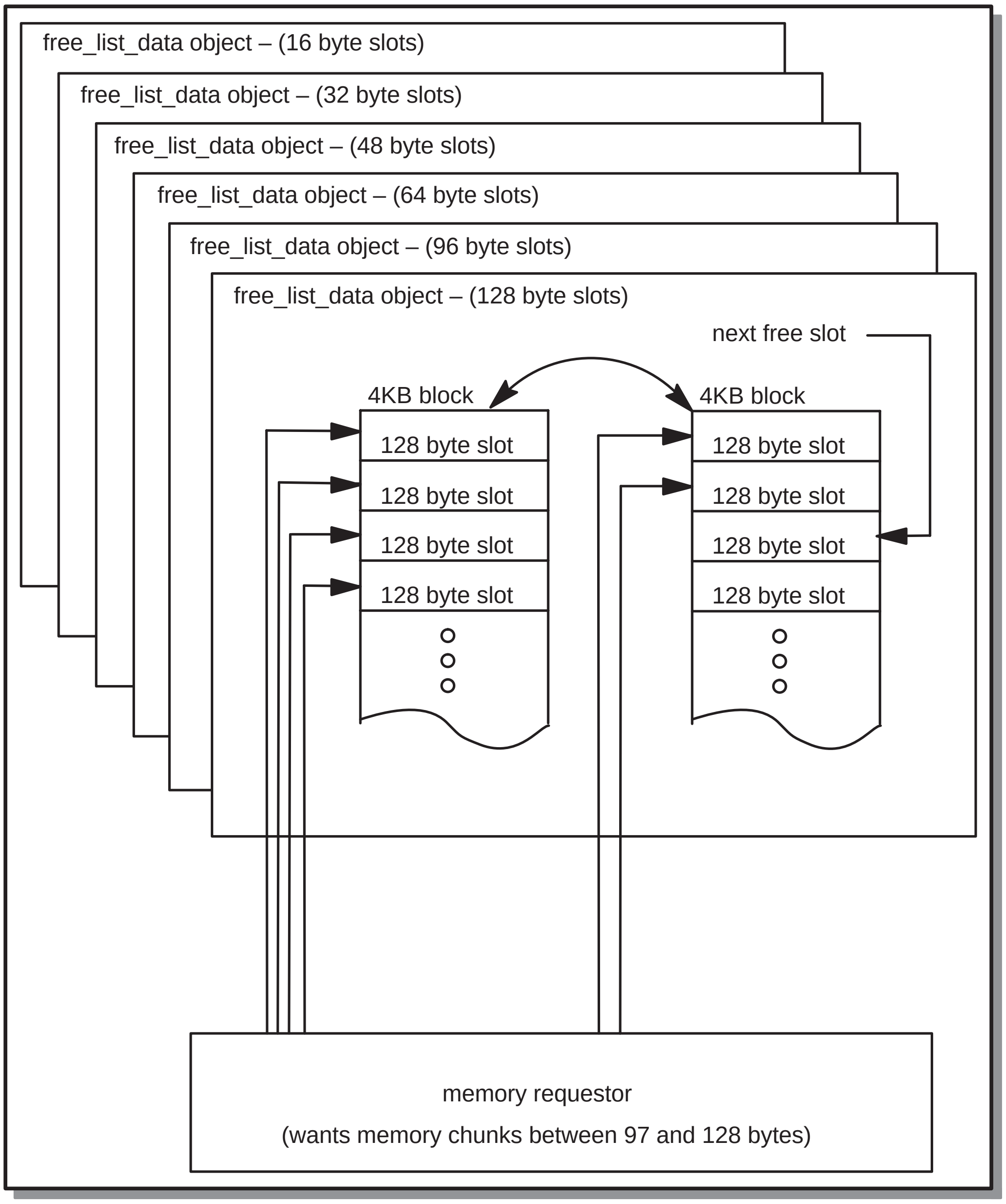
free_list_data object – (16 byte slots)
free_list_data object – (32 byte slots)
free_list_data object – (48 byte slots)
free_list_data object – (64 byte slots)
free_list_data object – (96 byte slots)
free_list_data object – (128 byte slots)
next free slot
4KB block
4KB block
128 byte slot
128 byte slot
128 byte slot
128 byte slot
128 byte slot
128 byte slot
128 byte slot
128 byte slot
memory requestor
(wants memory chunks between 97 and 128 bytes)

Figure 1-1. Free Lists

When the program frees the memory, the corresponding slot within the memory block is marked as unused. When all slots within a given block are marked as unused, the block can either be returned to the operating system or kept and reused. The application has three choices in dealing with unused memory blocks.

- *Unused blocks can be retained indefinitely. (default)* This leads to very fast execution by minimizing the number of operating system allocation requests, but creates a memory footprint that grows without shrinking until program termination.
- *Unused blocks can be automatically returned to the operating system.* This yields a small and dynamic memory footprint for ACIS, but may result in more (slower) allocations from the operating system as memory is requested.
- *The application can return unused blocks to the operating system when it chooses.* In this case unused blocks are retained and reused until the application specifically collapses the free lists to return unused memory to the operating system. For example, the application could keep all unused blocks while a model is read in and worked on, thus speeding execution, and then, when done with that model, collapse the free lists and return all now unused memory to the operating system.

Platform Defaults

Topic: *Memory Management

ACIS is shipped with the maximum level of memory management functionality available on the platform. At run–time, customers can modify the level of memory management used.

To determine the defaults and compiler directives for each platform, look in the build tool config files for the platform in the bldcfg directory. These files are named config.<platform>.

The release libraries are fully optimized. The release version will default to enable the freelists provided by the delivered system and will default to disable the auditing capabilities. The test applications bundled with our standard releases utilize the ACIS components as shipped.