Digital Equipment Corporation - Confidential and Proprietary
For Internal Use Only

Mica Working Design Document
Object Architecture

Revision 2.1

2-December-1987

Issued by:
Lou Perazzoli

digital™

TABLE OF CONTENTS

FIGURES

TABLES

Revision History

Date	Revision Number	Summary of Changes
18-Feb-1986	0.1	Original (Jim Kelly)
27-Feb-1986	0.1	Sent out for initial review (Jim Kelly)
28-Feb-1986	0.2	Add more detail, incorporate review changes. Primary changes include changes in terminology, making process level root object containers visible to descendants, allowing an object container to be transferred, adding in an "identifer" level, adding object driver description.(Jim Kelly)
21-Mar-1986	0.2	Sent out for second review.
19-May-1986	0.3	Reformatted to adhere to WDD structure. Incorporated changes from second review. Changes included changes in terminology, adding an additional process root container, changing the way transferability of an object container is determined, adding temporary object support, and various other minor changes.(Jim Kelly)
06-Jun-1986	0.3	Sent out for third review (Jim Kelly)
17-Jun-1986	0.4	Incorporated changed from third review. Delivered for incorporation into the WDD.
10-Jul-1986	1.0	Incorpated changes from WDD consistency review. The most significant of these changes included removal of the identifier level, definition of the name table structure and capabilities, removal of bonded objects apabilities, addition of alias objects,addition of temporary objects. (Jim Kelly)
02-Dec-1986	1.1	Attempted to simplify and unify. Major changes include the removal of acronyms, simplification of lock strategy, unification of name strategy, addition of logical names, addition of object allocation. (Lou Perazzoli)
28-Jan-1987	1.2	Removal of rooted containers and addition of directory containers. Added object services (e$ routines). (Lou Perazzoli)
20-Mar-1987	1.3	Remove alias, etc, add support for UNIX fork and exec. Added algorithums for object services. (Lou Perazzoli)
30-May-1987	1.4	Minor changes(Lou Perazzoli)
15-Sep-1987	2.0	Removal of fork and exec support for UNIX, minor fix-ups, added SIL definitions for object structures and quota section. (Lou Perazzoli)
24-Nov-1987	2.1	Minor changes. Changed E$ routines to OBJ$ routines and changed names to match the coding standard. (Lou Perazzoli)

CHAPTER 1

OBJECT ARCHITECTURE

1.1 Overview

1.1.1 Introduction

This chapter describes the software architecture of objects. It describes what objects are, and defines the data structures and operations necessary to support objects.

1.1.2 What is an Object?

Objects are abstract elements provided by an operating system that may be accessed by a user or a program. Typically, objects are defined in terms of the operations that may be performed upon them (for example, create, clear, set, get information, wait, delete) and their relationships to other objects. The reason for categorizing these elements as objects is to provide a single, standardized set of rules for creating, naming, protecting, accessing, and managing them. For example, each object has a unique ID value (called an object ID) which may be used to identify it. Objects at the job and process levels are only directly expressible by threads in that job or process.

1.1.3 Scope

It is important to understand that this chapter only defines the architecture of objects, not all object types. It is necessary that some objects or parts of objects be defined as part of this architecture.

1.1.4 Requirements and Goals

- Software development goals
  - — Provide an extensible, yet rigorous framework for the definition and manipulation of executive-controlled data structures.
  - — Maintain management consistency. The management of objects, in terms of actions taken to fulfill service requests, should be as object-type independent as possible. For example, standard routines and procedures can be established for determining whether access to an object should be granted.
  - — Provide new object definition support. It should be possible to add new object types to the system without having to modify existing system code. This means that the interface between the kernel/executive system software and objects must be well-defined, and that the kernel/executive need not have knowledge of the internals of all objects.
- Interface goals
  - — Provide consistent specification. The ways in which each object in the system may be specified by users should be minimized and kept consistent with the manner in which other objects are referenced.

— Provide consistent operations. There are some operations that apply to a set of objects within the system, such as wait. The definition of what these operations mean to each object should be kept simple and similar to their definition for other objects.

— Support level independence. Where possible, the operations that may be performed on an object type, and the behavior of that object, should not be dependent upon the level (system, job, or process) at which that object has been created. This allows applications to be developed in the relative safety of process and job levels before being moved to a more shareable level, with minimal change in behavior.

— Provide security and protection. The method of determining which objects a user may refer to, and which operations may be performed on those objects, should be the same for all objects. This is the basis for Mica security.

1.1.5 Functional Description

The object architecture runs in kernel mode at IPL 0. Through the use of mutexes, object architecture procedures can simultaneously execute on multiple processors.

The object architecture provides a framework for creating object-specific services. These services include creating, deleting, allocating, referencing, name translating, and getting information about objects. For example, the object service to create an event, and the object service to create a thread both invoke the same object architecture-defined routine to create the object.

The object architecture provides a hierarchical visibility structure for objects. When an object is created, it is placed at one of three levels: system, job, or process. Objects at the system level are visible to all threads on the system. Objects at the job level for a particular job are only visible to threads in that job. Objects at the process level for a particular process are only visible to threads in that process. For example, a thread cannot access an object that is at the process level for another process, because it cannot express an object ID for that object.

Each level can contain one or more object containers to catalog objects at that level. There are two types of object containers at the process level: Process-private object containers and display object containers. Objects stored in a process-private object container are only visible to the process with which the container is associated. Objects stored in a display object container are visible to the associated process, and any of its descendant processes.

Objects are referred to by object ID. If a program refers to an object name, this name must be translated to an object ID. An object name is unique within a container for each object type at each processor mode. When a user attempts to refer to an object using an object ID, the user's access rights are compared to the access rights associated with an object. If there is a match, the user may access the object.

An object may be allocated to a user, resource ID, job, process, or thread. This allows objects to be shared among restrictive classes of users.

An object ID is deleted when the object container holding the corresponding object is deleted, or when the object ID is explicitly deleted. The object itself, however, is not deleted until the ID is deleted, and there are no outstanding references to the object.

Table 1–1 summarizes key object architecture terms and components.

Table 1–1: Object Architecture: Terms and Definitions

Object Identification and Names		
Term	Type	Definition
Object ID	64-bit Value	Used to refer to an object.

Table 1–1 (Cont.): Object Architecture: Terms and Definitions

Object Identification and Names

Term	Type	Definition
Principal Object ID	Object ID	Associated with an object at object creation. An object has exactly one principal object ID.
Reference Object ID	Object ID	Optionally associated with an object. An object may have zero, one, or more reference IDs.
Object Name	Character String	Together with type and mode, an object name can be translated to an object ID. The combination of object type, mode, and name string is unique within a single object container.
Object Name Table	Data Structure	Tracks both logical names and object names within an object container. When an object container is created, a name table is also allocated, and that address is stored in the object container's body.

Object Hierarchy

Term	Definition	Description
Object Container	Object Type	Objects of this type contain pointers to other objects. They are used to organize large numbers of objects.
Object Level	–	Indicates the scope of visibility of an object container.
System Level	Object Level	Objects at this level are potentially accessible to all processes on the system.
Job Level	Object Level	Objects at this level are potentially accessible to all processes in a given job.
Process Level	Object Level	Objects at this level are potentially accessible to all threads in a given process. Containers at this level can be ether display or private.
Display Object Container	Object Container	Objects in such containers are accessible to a given process and all of its descendants.
Private Object Container	Object Container	Objects in such containers are accessible only to a given process, and not to its descendants.
Container Directory	Data Structure	Used to organize large numbers of object containers. All threads have the same system container directory. All threads in a job have the same job container directory. All threads in a process have the same process container directory.
Object Header	Data Structure	Fixed-format data structure that contains object type-independent data. This header is used by the executive without necessarily knowing the type of object it is accessing.
Object Body	Data Structure	A data structure that is specific to an object type.

Table 1–1 (Cont.): Object Architecture: Terms and Definitions

Object Type

Term	Definition	Description
Object Type	–	Object type determines what operations can be performed on an object.
Object Type Descriptor (OTD)	Data Structure	Describes what operations are supported for what object types. There is one OTD for each object type.

Miscellaneous

Term	Definition	Description
Object Service Routines	System Routines	Implement operations that can be performed on objects. Some object service routines are particular to a certain type of object; others are supported across all object types.
Object Allocation Block	Data Structure	Contains information about object allocation.

1.1.6 Object-Related Operations

The following types of operations can be performed on most types of objects:

- Creating an object
- Protecting an object
- Translating an object ID
- Deleting an object
- Creating references to an object
- Making a temporary object
- Marking a new object as temporary
- Allocating an object
- Deallocating an object
- Getting information about an object
- Changing the name of an object

1.1.7 Summary of Proposed Changes to Object Architecture

We propose that the following changes be made to the existing object architecture chapter:

- Removal of all ULTRIX dependencies. These include clone procedures and executive actions.
- Addition of Pillar definition records.
- Deduction of quota rules.

1.2 Functional Interface and Description

An *object* is a set of data structures representing an abstraction. Processes and events are examples of objects. In the case of an event object, the object represents the abstraction of an event that either has or has not taken place.

An object consists of a standard data structure called the *object header*, described in Section 1.4, and an object type-dependent data structure called the *object body*.

1.2.1 When Should an Element in the Executive be an Object?

An element should be an object when it:

- Needs a name for sharing.
- Needs to be referenced by user-mode software, and there are multiple instances of these elements.
- Needs to be protected via ACLs. Exceptions to this include structures that are exported by an RPC, such as job controller queues.
- Can be allocated.

No object can be dependent on another object such that there is an order dependence during object ID deletion. This allows various rundown operations to delete object IDs in any order, yet the rundown proceeds efficiently.

1.2.2 Object IDs

When an object is created, it is assigned a 64-bit value called its *object ID*. This value provides the fastest means for a user-mode routine to identify and access the object.

When an object is created, the ID returned is called the *principal ID*. Every object has only one principal ID. An object may also have one or more reference IDs which also point to the object. Reference IDs are discussed in Section 1.7.9.

The principal ID of an object that is accessible by two processes is the same in both processes. This extends to objects at all levels (including those in ancestors' process-display object containers, described in Section 1.5.9). This property allows IDs of shared objects to be communicated between cooperating threads.

1.2.3 Object ID Format

Object IDs are used to uniquely identify objects within the system. Object IDs are not addresses of objects, but values that may be used to generate the addresses of objects.

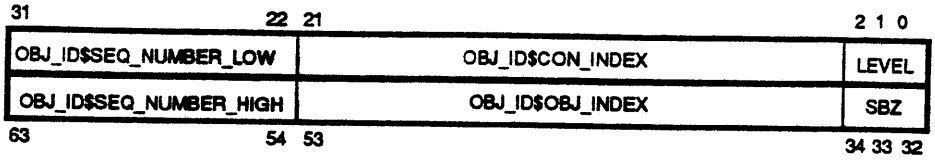
Object IDs are comprised of the following fields:

- Sequence number (*obj_id$seq_number_low*, *obj_id$seq_number_high*)
- Level (*obj_id$level*)
- Object container index (*obj_id$obj_index*)
- Container directory index (*obj_id$con_index*)

The format of an object ID is shown in Figure 1–1.

Object ID value zero (0) is reserved for use as an invalid value.

Figure 1–1: Object ID Format



1.2.4 Object ID Sequence Numbers

Three object ID fields are used to locate the address of the object.They are the level field, the object container index field, and the container directory index field. In addition, the sequence number high and low are used to help catch programming errors. Such an error might occur if a program used an object ID after the corresponding object had been deleted, and another object had been created with the same ID as the previously deleted object.

For example, consider a program with the following error. First, the program creates a FIZBOT object (which receives a corresponding object ID). Sometime later, the program deletes the FIZBOT object and creates a WALZOL object. Suppose that, due to a programming error, an operation on the FIZBOT object is attempted (such as a wait operation). If the WALZOL object had coincidentally received the same object ID that had been used for the FIZBOT object, a wait on the wrong object might go undetected.

To catch this type of error, each time an object ID is created, the sequence number fields (two 10 bit fields) are different each time an object ID is assigned to the same "slot".

When an object ID is created, the old sequence high field, which was saved, is incremented and reset to zero in the overflow case. In our example above, the improper use of the FIZBOT object ID would be detected and flagged as an error (no such object). Notice that the sequence high field cycles every 1024 allocations of an object ID.

The sequence low field receives a random value.

The use of sequence numbers does not totally eliminate the problem; it just reduces it to an extremely small probability for most programming situations.

1.2.5 Object Containers

Since programs are likely to have many objects at any point in time, it is desirable to be able to group objects together in some logical manner. For this purpose, a type of object called an *object container* is defined. When creating an object, it is necessary to specify the ID of the object container in which a pointer to the newly created object is to be stored.

From the executive's point of view, object containers are used when translating an object ID into a pointer to the object.

1.2.6 Object Levels

There are three levels of object visibility: process, job and system. The principal object ID level field determines the visibility of an object. The software process control block (SWPCB) contains level directors for the object levels. For each level, there is a pointer to the container directory corresponding to that level.

1.2.6.1 System Level

The purpose of the system level is to provide a place to keep objects shared by multiple jobs. Implementation of the group concept supported by VMS uses this level of sharing, by creating an object container at the system level that is only accessible to threads with the proper group identifier.

Objects at this level are potentially accessible by all processes in the system. If access to an object at this level is to be restricted, then access protection must be explicitly assigned.

1.2.6.2 System Level Director Mutex

A system-wide mutex exists for synchronizing access to the system level containers. A pointer to this mutex is located in each software process control block. This mutex is acquired for object ID translations at the system level and for other system-level operations, such as Delete Object ID.

1.2.6.3 Job Level

The purpose of the job level is to allow objects to be shared by all processes within a single job. If access to an object at this level is to be restricted so that only some processes within a job may access it, then access protection must be explicitly assigned.

1.2.6.4 Job Level Director Mutex

A job-wide mutex exists for synchronizing access to the job level containers. A pointer to this mutex is located in each process control block. This mutex is acquired for object ID translations at the job level, and for other job level operations.

1.2.6.5 Process Level

The process level has two types of containers, display and private.

The purpose of process-private object containers is to provide a location for objects that are not accessible to any other process. No access protection needs to be assigned to objects in this type of object container. This simplifies the programming effort, and also provides the fastest possible object access.

The purpose of process-display object containers is to provide a location for objects that are accessible to the associated process, and to all descendant processes of that process. Objects in this object container do not typically require any access protection. Note, however, that if access protection is assigned to objects at the process level, it is checked at the time of each access.

It is important to note that the process-display object containers are only accessible to a process and its descendant processes.

1.2.6.6 Process Level Director Mutex

A job-wide mutex exists for synchronizing access to the process level containers. A pointer to this mutex is located in each process control block. This mutex is acquired for object ID translations at the process level, and for other process-level operations.

This mutex must be job-wide to allow proper synchronization on display containers which are shared among a process and its descendants.

1.2.7 Container Directory Index Field of the Object ID

The container directory index field is used as an index into the object array of the container directory. This yields a pointer to the object container, which contains a pointer to the object represented by the object ID.

1.2.8 Container Directory

For each level there is a corresponding container directory. All threads share the same system container directory. All threads within the same job share the same job container directory. All threads within a process shares the same process container directory.

The total number of container directories within the system at any given time is equal to the number of jobs plus the number of processes plus one (for the system level container directory).

The container directory provides a structure containing pointers to all the object containers within that level. It also provides a method of naming object containers.

1.2.8.1 Object Index Field of the Object ID

The object index field is used as an index into the object array of the object container located by the directory index. This yields a pointer to the object header of the object represented by the object ID.

1.2.9 Expressibility of Object IDs

One characteristic of the architecture is that threads are not able to directly express the ID of all objects. Here, the term "directly express" means to generate an ID value that corresponds to an object.

All threads are able to directly express the ID of all objects at the system level. However, objects at the job and process levels are only directly expressible by threads in that job or process. For example, there is no object ID value a thread in job X can use to directly refer to an object at the job or process level in job Y. Also, a thread in process Q cannot directly refer to process level objects in any other process, unless the object is in a process-display object container of an ancestor process (process-display object containers are discussed in Section 1.2.6.5).

1.2.10 Shareability of Object IDs

If two threads can both express the principal ID of an object, then those ID values are the same value. This allows cooperating threads in different processes or jobs to communicate object IDs to one another.

Notice that this property even extends to objects in ancestor process-display object containers. This allows a thread in one process to communicate an object ID of an object in its process-display object container to a thread in a descendant process (process-display object containers are discussed in Section 1.2.6.5).

1.2.11 Translation of an Object ID to an Object Header Address

In order to translate an object ID to the address of the associated object, Mica performs the following steps:

1. Uses the level number as an index into the process control block to locate the corresponding level directory mutex.
2. Acquires the directory mutex.
3. Uses the level number as an index into the process control block to locate the corresponding container directory.
4. Compares the container directory index field to the size of the table to ensure that index is within the object array.
5. Uses the container directory index field to index into the object array.
6. Checks the object array element to ensure it contains the address of an object header.
7. Uses the resulting system address to locate the object container header for the object container.

8. Compares the object index field to the size of the table to ensure that index is within the object array.
9. Uses the object index field to index into the object array. Checks the resulting value to ensure that it is the address of an object.
10. Compares both sequence number fields in the object header to the sequence number fields in the object ID. To do this, Mica checks the address of the object header, which is found in the object array. If they are not identical, then that object ID is not valid.
11. Increments the pointer count field in the object header so the object's storage cannot be deallocated while a pointer to the object is held.
12. Releases the directory mutex, and returns a pointer to the object body to the caller.

The routine *obj$reference_object_by_id* described in Section 1.7.6 provides the mechanism to translate an object ID to the address of the object header. All routines within the executive use *obj$reference_object_by_id* for translating object IDs.

Figure 1–2 indicates how the structures fit together.

1.2.12 Container Directory IDs

It is necessary for programs to find container directories for each level. Since no consumer of the object architecture may have any knowledge about the construction of an object ID, programs call a system service that returns the object ID of the desired container directory.

The container directories are named as follows:

- The system level container directory is named *exec$system_container_directory*.
- The job level container directory is named *exec$job_container_directory*.
- The process level container directory is named *exec$process_container_directory*.

The *exec$translate_object_name* service may be used to obtain the object ID of a container directory by translating one of these three names. The caller must specify the container directory name for both the object name to locate and the container directory in which to locate the name.

1.2.13 Object Access By ID Only

One guideline to follow in system software design is that user-mode access to objects is only allowed by object ID, and is not allowed by name. If the object must be located via a name, the translation from name to ID is performed in a separate call preceding the object access call. This simplifies the coding of the object service routines, since they do not have to do any name translations or deal with logical names.

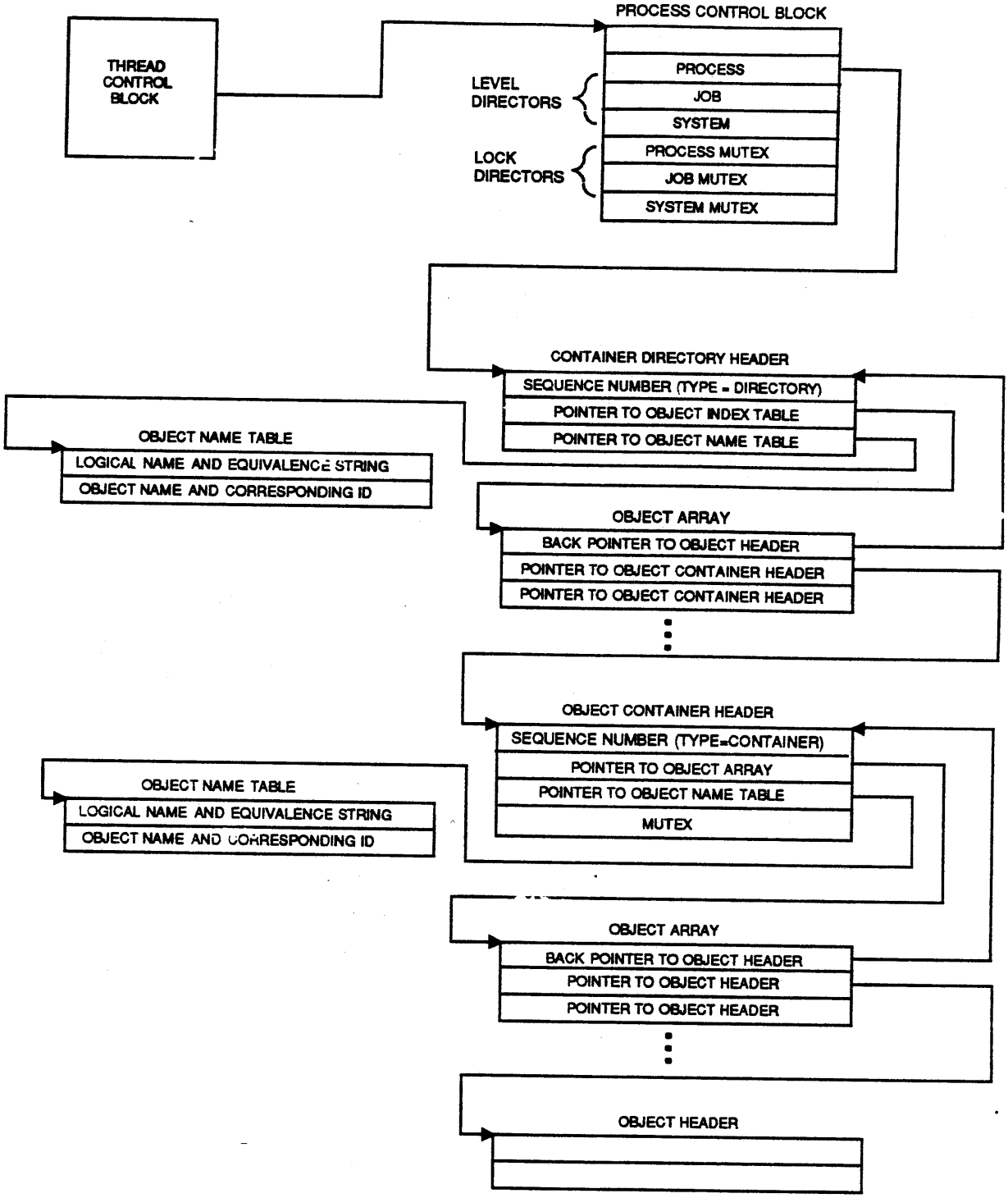
1.2.14 Object Service Routines

Each type of object has an associated set of services called *object service routines*. The service routines for a particular object type perform the operations supported by that object type.

For example, a FRAMITZ object type may define the operations:

- create_framitz
- get_framitz_information
- clear_framitz
- juggle_framitz

Figure 1–2: The Big Picture



- read_framitz
- close_framitz

Each of these operations is represented by a separate object service routine. These object service routines can be added to the system dynamically, but only as a complete group.

Every set of object service routines must supply a method for creating the object and providing information about the object. These services are named *create*_object_type, and *get*_object_type_*information*. Each set of object service routines must also implement routines to provide system defined object type-specific operations.

These operations are:

- Remove
- Delete
- Shutdown
- Allocate
- Deallocate

These operations are used by the executive, and, in some cases, by the object's service routines. The availability and location of these functions, as well as some status and control information, are provided in object type descriptors (OTDs).

1.3 Object Type Descriptor (OTD)

There is a single object type descriptor (OTD) object for each object type defined by the system. All OTD objects are created in a single system container.

An OTD object, hereafter referred to as just OTD, contains general information about an object type, not a particular instance of an object type. For example, an OTD indicates whether an object type supports the notion of wait. Anything an OTD specifies about an object type applies to all instances of that object type.

Part of the information in an OTD is the location of a number of routines that may be called by the executive. These routines have standard definitions across all object types but require object type-specific processing. These routines are located by the executive via standard offsets into the OTD. The operations are listed in the previous section.

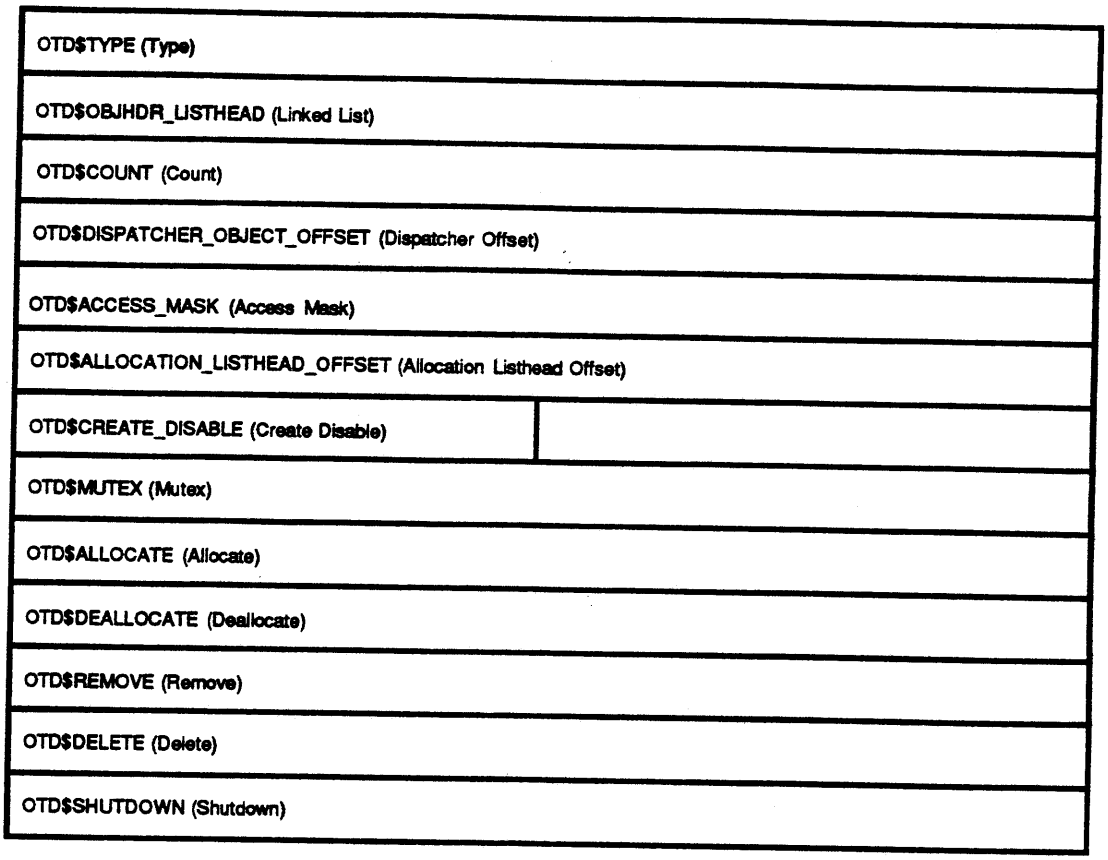
Users of an object type do not need to know anything about the routines pointed to by the OTD. The designer of an object type, however, must specify and implement them.

1.3.1 Object Type Descriptor Body Format

```
e$object_type_descriptor : RECORD
    otd$type : e$object_type;
    otd$objhdr_listhead : e$linked_list;
    otd$count : integer;
    otd$dispatcher_object_offset : integer;
    otd$access_mask : POINTER e$access_mask;
    odt$allocation_listhead_offset : integer;
    otd$create_disable : bit; ! 0 - enable, 1 - disable
    otd$mutex : k$dispatcher_object (mutex);
    otd$allocate : obj$obj_allocate_procedure;
    otd$deallocate : obj$obj_deallocate_procedure;
    otd$remove : obj$obj_remove_procedure;
    otd$delete : obj$obj_delete_procedure;
    otd$shutdown : obj$obj_shutdown_procedure;
END RECORD;
```

All fields of the OTD have a standard definition. This allows the executive to use the information in an OTD without having detailed knowledge of the corresponding object type.

Figure 1–3: Object Type Descriptor (OTD) Format



The purpose of each of these fields is described below.

1.3.1.1 otd$type Field (Static)

This field contains a value indicating the type of object the OTD represents.

1.3.1.2 otd$objhdr_listhead Field (Dynamic)

This field contains the forward link to the first object header of this type, and the backward link to the last object header of this type. These links are used for consistency checking within the object architecture.

1.3.1.3 otd$count Field (Dynamic)

This field contains a count of the number of objects of the corresponding type that currently exist in the system. Modification of this field must be done using the RMALI instruction.

1.3.1.4 otd$dispatcher_object_offset Field

When nonzero, this object type supports the notion of wait. The value in this field is the offset of the dispatcher object (dispatcher objects are described in Chapter 6, The Kernel) from the start of the object body. Thus, when a wait operation is performed on the object, the executive adds the field contents to the address of the object header and issues a kernel wait operation on the dispatcher object.

The dispatcher objects that support waiting are:

- Events
- Semaphore
- Timer
- Thread
- Queue

These objects are created by providing the necessary structure in the object body and calling the kernel routine *k$initialize_xxxxx*, where xxxxx is the type of dispatcher object to be initialized. Once initialized, these objects can then be manipulated by using other kernel routines. For more details on the kernel support provided, see Chapter 6, The Kernel.

Any object type which has a non zero dispatcher field must allocate the object body from nonpaged pool. The wait routines which accept object types with non-zero dispatcher offsets call the kernel wait routines which operate at IPL 2 and cannot take page faults.

1.3.1.5 otd$access_mask Field (Static)

This field points to the supported access mask for use by the security routines in access validation. For more information, see the Chapter 11, Security and Privileges.

1.3.1.6 otd$allocation_listhead_offset Field

When nonzero, this object type may have objects allocated to it. For example, thread, process and job objects all support the notion of object allocation. Object allocation is described in Section 1.8.

1.3.1.7 otd$create_disable Field (Dynamic)

This flag may be set by the executive to prevent additional objects of this type from being created. It can be used to shut down the system in an orderly fashion.

Once set, this field may not be cleared. Therefore, access does not have to be interlocked. This flag is set when it contains a nonzero value.

1.3.1.8 otd$mutex Field

Provides synchronization for creation, deletion, and state changes among objects within a type.

1.3.1.9 otd$allocate Field

This field points to an allocate procedure. The allocate procedure is called in kernel mode at IPL 0 with the allocation mutex acquired. The address of the object body to be allocated and the allocation type are passed as input parameters.

The allocate procedure has the following type declaration:

```
obj$object_allocate_procedure :
    PROCEDURE (
        IN object_body : POINTER anytype CONFORM;
        IN allocation_object_hdr : POINTER e$object_header;
        );
```

The procedure *obj$null_allocate_procedure* is provided for use by object types which do not have an allocate procedure.

1.3.1.10 otd$deallocate Field

This field points to a deallocate procedure. The deallocate procedure is called in kernel mode at IPL 0 with the allocation mutex acquired. The address of the object body to be deallocated is passed as an input parameter.

The deallocate procedure has the following type declaration:

```
obj$object_deallocate_procedure :
    PROCEDURE (
        IN object_body : POINTER anytype CONFORM;
        IN allocation_object_hdr : POINTER e$object_header;
        );
```

The procedure *obj$null_deallocate_procedure* is provided for use by object types which do not have an deallocate procedure.

1.3.1.11 otd$remove Field

This field points to a remove procedure. The remove procedure is called when an object's object ID count field is decremented to 0. It is called in kernel mode, at IPL zero. It is passed two arguments, the address of the object body and the mode (user or kernel).

The remove procedure allows the object type-specific procedure to perform any necessary actions now that the ID of the object is removed. Once object ID count field is zero, it is impossible for a user to translate an object ID which points to this object.

The remove procedure has the following type declaration:

```
obj$obj_remove_procedure :
    PROCEDURE (
        IN object_body : POINTER anytype CONFORM;
        IN access_mode : k$processor_mode;
        );
```

The procedure *obj$null_remove_procedure* is provided for use by object types which do not have a remove procedure.

1.3.1.12 otd$delete Field

This field points to a delete procedure. The delete procedure is called when an object's pointer count field is decremented to zero. It is called in kernel mode at IPL zero. It is passed the address of the object body.

The delete procedure is responsible for manipulating object type-dependent data structures and deallocating the storage allocated for the object body extensions.

The delete procedure has the following type declaration:

```
obj$obj_delete_procedure :
    PROCEDURE (
        IN object_body : POINTER anytype CONFORM;
        );
```

The procedure *obj$null_delete_procedure* is provided for use by object types which do not have a delete procedure.

1.3.1.13 otd$shutdown Field

This field points to a shutdown procedure. The shutdown procedure is called once when an object type is permanently removed from the system (presumably at system shutdown time). It is called by the executive in kernel mode at IPL 0 with ASTs enabled. The purpose of this routine is to provide a way to perform object type-specific shutdown operations. Among other things, this provides the opportunity to dump any statistics that may have been gathered relating to the object type.

The parameters passed to this routine include the address of the OTD and the address of a callback routine that can be used to output information to a shutdown log file. This routine must be provided even if it simply returns without performing any actions.

The shutdown procedure has the following type declaration:

```
obj$obj_shutdown_procedure :
    PROCEDURE (
        IN otd_hdr : POINTER e$object_header;
        IN log_procedure : e$log_procedure;
        );
```

The procedure *obj$null_shutdown_procedure* is provided for use by object types which do not have a shutdown procedure. procedure.

1.4 Object Header

The object header is a fixed-format data structure that contains object type-independent data. This header is used by the executive without necessarily knowing the type of object it is accessing.

```
e$object_header : RECORD
    objhdr$pointer_count : integer;
    objhdr$object_id_count : integer;

    objhdr$type : e$object_type;
    objhdr$otd : POINTER e$object_header;
    objhdr$link : e$linked_list;

    objhdr$container : POINTER e$object_header;
    objhdr$level : e$level;
    objhdr$index : e$index;
    objhdr$seq_number_low : e$seq_number;
    objhdr$seq_number_high : e$seq_number;
```

```
        objhdr$dispatcher_object : POINTER anytype CONFORM; !# POINTER k$dispatcher_object;
        objhdr$name : POINTER e$object_name_block;
        objhdr$owner : e$identifier;
        objhdr$acl : POINTER e$access_control_list;
        objhdr$allocation_block : POINTER e$object_allocation_block;
        objhdr$access_mode : k$processor_mode;
        objhdr$transfer_flag : bit; ! 0 - transfer not allowed
                                    ! 1 - transfer allowed
        objhdr$reference_inhibit_flag : bit; ! 0 - reference ids allowed
                                             ! 1 - reference ids not allowed
        objhdr$temporary_flag : bit; ! 0 - permanent
                                     ! 1 - temporary
        objhdr$temporary_operation : bit;    ! 0 - make temporary
                                             ! 1 - mark temporary
    END RECORD;
```

1.4.1 Object Header Data Structure

Each of the fields in the object header are defined to be either static or dynamic. If a field is defined as static, its value is established during object creation, and remains constant for the life of the object. Static fields may be accessed any time the pointer count field has a nonzero value (see the description of the *objhdr$pointer_count* field in Section 1.4.1.1). However, if a field is defined to be dynamic, it is subject to modification. Access to dynamic fields is controlled by a protocol that is specific to each field. The specific protocol indicating when each dynamic field may be accessed is defined with the description of the field.

1.4.1.1 objhdr$pointer_count Field (Dynamic)

Whenever an access to an object will span a period of time that can not be protected by use of mutex locks on appropriate data structures, the pointer count field must be incremented. In particular, an address of an object may not be used to regain access to the object unless the pointer count has been previously incremented.

The pointer count represents the number of pointers that have been taken out on the object, plus one for a nonzero object ID count. When an object ID is translated by an object service routine, the pointer count is incremented by one. The pointer count for an object signifies the number of reasons the storage for an object should not be deallocated.

To increment this field, the directory-level mutex must first be held (see Section 1.5.3.4, below). This is necessary to avoid race conditions with the object ID being deleted.

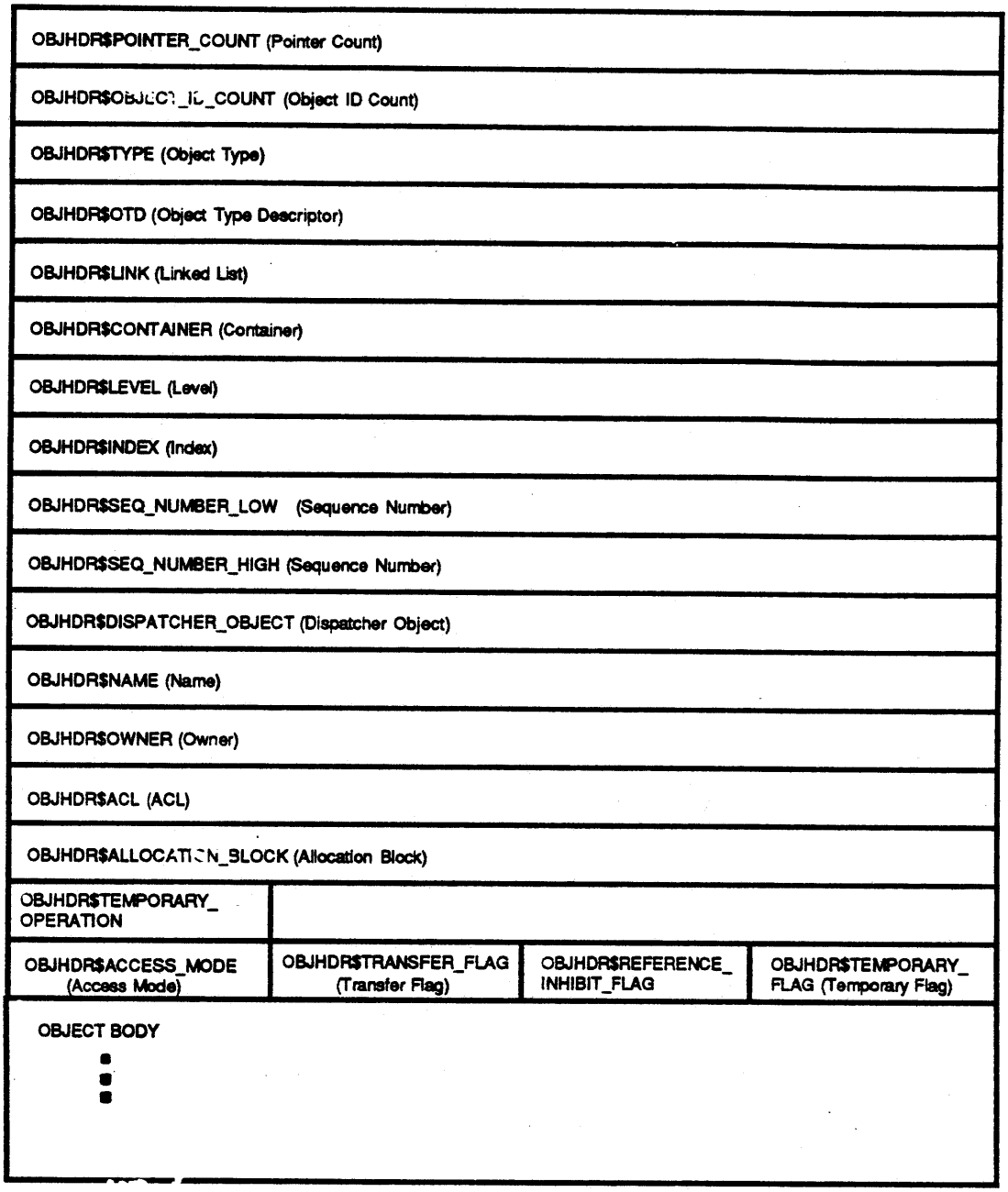
When an object ID, object container pointer, or reference object ID is deleted, the object ID count of the principal object is decremented. If the resultant object ID count is zero, the object type-specific remove routine is called to initiate any object specific removal action such as canceling I/O. A zero object ID count also causes the pointer count to be decremented, as does the dereferencing of a pointer. When the pointer count of an object reaches zero, the object type-specific delete routine is called, and the object header storage is deallocated.

Note, if the object ID count is zero then no object container has a pointer to that object header.

One of the major goals of the lock strategy for objects is to allow the pointer count to be decremented (using the RMALI instruction) without having to hold a lock to do so. This allows low overhead dereferencing of objects on such operations as I/O, wait, and the deleting of a reference object.

If the pointer count is zero and the object ID count is nonzero, a bug check is issued.

Figure 1–4: Object Header Format



1.4.1.2 objhdr$object_id_count Field (Dynamic)

The object ID count represents the number of object IDs or container directory index table pointers that can be used to refer to the object. For all objects except container objects, this is the number of object IDs that refer to the object. For container objects, this is the number of container directories that can refer to this object container (note that the object ID itself is not counted). Hence, for object containers, the object ID count is one unless the container is a display container.

The object ID count signifies the number of reasons that the remove action for an object should not be invoked.

If an object is marked as temporary (as discussed in Section 1.4.1.18, below), then it must have at least one reference to it.

To increment this field, the mutex field of the object container in which the object resides and the directory-level mutex must first be acquired (see Section 1.5.3.4, below). This is necessary to ensure that the object cannot be deleted while another reference is being established.

Note that this field is never directly incremented or decremented; the executive routines for object support operate upon this field.

1.4.1.3 objhdr$type Field (Static)

This field contains the integer value of the object type. When an object ID is translated to an object header, the type field is compared to the desired object type.

1.4.1.4 objhdr$otd Field (Static)

The object type descriptor field contains a pointer to the object type descriptor (OTD) for this particular object type. The executive and the object type's service routines use this field to use information or routines associated with the OTD. Note that there is exactly one OTD per object type defined in the system.

The OTD is described in detail in Section 1.3.1.

1.4.1.5 objhdr$link Field (Dynamic)

This field contains the forward link to the next object header of this type, as well as the backward link to the previous object header of this type. It is used for debugging purposes.

1.4.1.6 objhdr$container Field (Dynamic)

The container field is a pointer to the object container in which the object resides.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in the assignment of a new object ID. This field is zeroed when the principal object ID is removed from its object container.

This field may only be accessed by a thread holding both the directory level mutex and the mutex within the object container.

1.4.1.7 objhdr$level Field (Dynamic)

The level (system, job, process) at which the object's principal ID exists. This field is unaffected if the principal ID is deleted.

1.4.1.8 objhdr$index Field (Dynamic)

This field is the index into the object array of the container object specified in the container field for this object.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in a new object ID being assigned. This field may only be accessed by a thread holding both the directory-level mutex and the mutex within the object container.

1.4.1.9 objhdr$seq_number_low and objhdr$seq_number_high Fields (Dynamic)

When an object ID is translated into the address of an object, the value in the object ID's sequence field is compared to the value in the object header's sequence field. If they are not identical, then the object ID has been reallocated, and the one specified for translation is no longer valid.

This field is dynamic only because the object may be transferred to a new object container. Such a transfer would result in the assignment of a new object ID. This field may only be accessed by a thread holding both the directory-level mutex, and the mutex within the object container.

Note that it is possible to construct the object ID given an object header. Two parts of the object ID are in the object header, the sequence number and the object index. The container field points to the object header of a container. The index field in that container is the directory index. The level value in the container directory body is the level portion of the ID.

1.4.1.10 objhdr$dispatcher_object Field (Static)

If the object type allows wait operations this field points to the dispatcher object to be used in the kernel wait operation. If the object type does not support the notion of wait, this field has the value nil.

1.4.1.11 objhdr$name Field (Dynamic)

This field is used to associate a name string with an object. If the object has an associated name, this field contains a pointer to an name definition block which contains the name. Otherwise, this field has a value of zero.

When the object ID is deleted, the associated name, if any, must also be deleted.

The level directory, the object container, and the type-specific mutex must all be acquired in order to modify this field.

See Section 1.5.5 for a complete description of how object names are managed.

1.4.1.12 objhdr$owner Field (Static)

The identifier of the owner of the object. This field is used by the security access validation routines. For more information see Chapter 11, Security and Privileges.

1.4.1.13 objhdr$acl Field (Dynamic)

The format and use of this field is defined in Chapter 11, Security and Privileges.

This field is a pointer to the ACL for the object. A value of nil indicates that no ACL exists for the object.

1.4.1.14 objhdr$allocation_block (Dynamic)

The field is used by the object translation routines to ensure that the thread has access to the object. In order to translate an object ID to an object header, the requesting thread must have access to the object as determined by the ACL, and must be in the allocation class of the object. Object allocation and allocation classes are discussed in Section 1.8.

The allocation mutex and the type mutex must be acquired in order to modify this field.

1.4.1.15 objhdr$access_mode Field (Static)

This field contains the owner mode (kernel or user) of the object. Kernel-mode objects may not be created in user owned containers. Mode is also used to verify access to the object by the access validation procedures as described in Chapter 11, Security and Privileges.

1.4.1.16 objhdr$transfer_action Field (Static)

When clear (false) this flag indicates that the object cannot be transferred to another container. The transfer action field state is determined at object creation from the transfer state flag in the OTD. This flag is examined when an attempt is made to transfer an object using the *exec$make_temporary* service or job/process creation services.

Transfer action is discussed in Section 1.7.8.

1.4.1.17 objhdr$reference_inhibit Field (Dynamic)

When set (true) an attempt to create a reference ID to this object results in a failure.

1.4.1.18 objhdr$temporary_flag Field (Dynamic)

When set, this flag indicates the object is a temporary object. A temporary object has the property that when all reference objects to the object are deleted, then the object itself is also deleted. Therefore, when the object ID count field is decremented to one, this field is checked. If it is set, then object deletion may be invoked. See Section 1.7.7 for more information.

The level directory, the object container, and the type-specific mutex must all be acquired in order to modify this field.

1.4.1.19 objhdr$temporary_operation Field (Dynamic)

This flag describes the type of operation that was used (make temporary or mark temporary) to make the object temporary. This flag is only meaningful when the *objhdr$temporary_flag* is set.

1.4.2 object_body Record

The purpose and use of this record is left to the designer of a particular object type.

The *object_body* record is used to store object type information. The format and meaning of fields within this record are entirely the responsibility of the object's designer. It is also up to the designer of the object type to define the protocols for accessing various fields within this record.

In many cases, the *object_body* record can be allocated in a single block (with the object header) and is sufficient to hold all the object type-specific information associated with an object. Certainly this is true for simple objects like events. For more complex objects, like object containers, it may be necessary to have other data structures associated with the object.

The content and management of these additional data structures is entirely the responsibility of the object's designer. They must be connected in some fashion to the *object_body* record so that they are deleted when the object is deleted.

Object bodies are allocated from paged or nonpaged pool. If the object body contains any kernel dispatcher objects or kernel control objects it must be allocated from nonpaged pool.

Object bodies should not contain the object IDs of any objects. If a need exists to refer to another object a pointer to that object's body should be used. This prevents dependencies on other object IDs being valid.

1.5 Object Container Data Structures

Object containers are objects. Therefore, their primary data structures are:

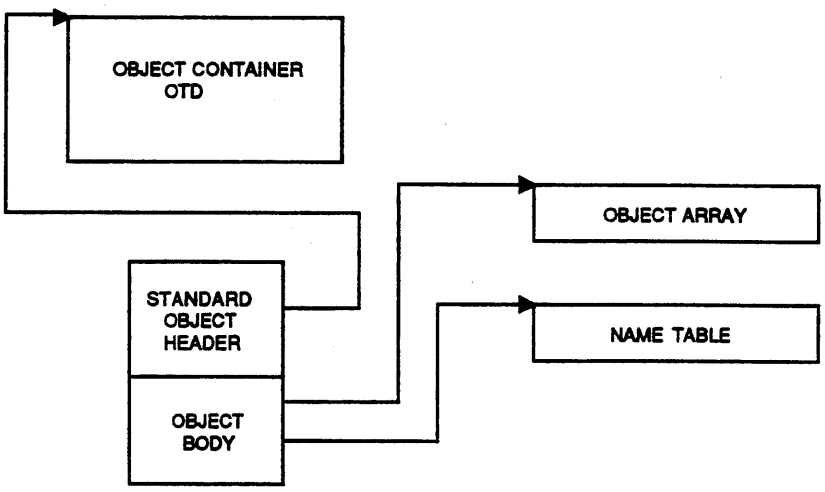
- Object container object type descriptor (OTD)
- Object container object header
- Object container body

To track objects and their associated names, object container bodies have two primary associated data structures. These data structures are:

- Object array
- Name table

Figure 1–5 illustrates the relationships between these object container data structures.

Figure 1–5: Object Container Data Structure Relationships



In addition to these data structures, object names are stored in name definition blocks.

\The exact data structures used to support object names, and is subject to change following performance analysis yet to be performed. The algorithms for manipulating the ultimate data structures will have to be implemented to match the data structures, and so, are also subject to change.\

The aspects of the current design that are not expected to change are:

- The name field of the object header points to a data structure containing the case-sensitive name of the object. This pointer provides the link necessary to delete the object name when the object is deleted.
- A field in the object container body (currently called the name table field) is used to locate the name management data structures for name assignment and translation purposes.

1.5.1 Container Directories as Compared with Object Containers

Container directories are similar in structure to object containers. The object header is the same, except for type, and the object body is nearly identical. The only differences are:

- Container directories do not have a mutex in their object bodies.
- Only container directories have a display index field.

There are some restrictions placed on the container directory that are enforced by the executive object manipulation routines. They are:

- The only objects that can be pointed to by (contained within) a container directory object are object containers.
- The only names that can be stored in a container directory are the names of the object containers that are pointed to by the container directory and logical names.

1.5.1.1 Object Container OTD Remove Procedure

When an object container is deleted, the OTD remove procedure removes the ID of each object within that container by calling the routine *obj$remove_obj_from_container*.

1.5.2 Object Container Object Header

The fields of the object container's object header are initialized and used as defined in Section 1.4. Note that the object container is an object, and is pointed to by the first pointer in the object index table.

Also, the object index field of the object header has a value of zero for a container directory. Object containers contain the value corresponding to the object index array element in the container directory, which points to this object container.

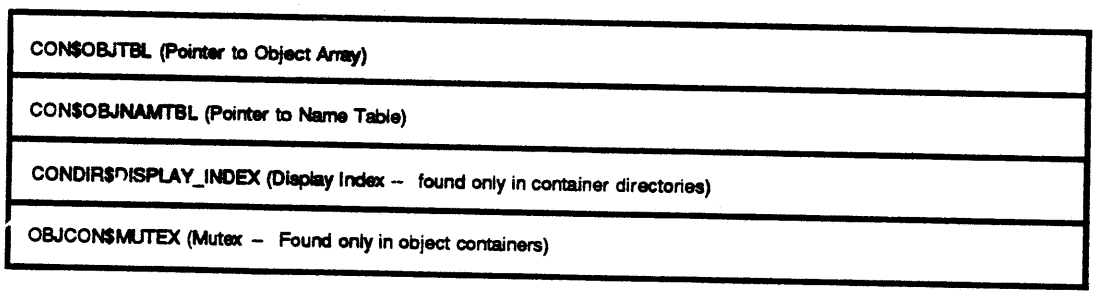
The following restrictions exist for object containers:

- An object container (or container directory) cannot be allocated.
- An object container (or container directory) cannot be temporary.

1.5.3 Object Container Body

The format of the object container's body is shown in Figure 1–6.

Figure 1–6: Object Container Body Data Structure



The purpose of each of these fields is described below.

The object array field is a pointer to the object container's object array. The object array is described in Section 1.5.4, below.

Note that the first object pointer of this table points to the object header for this container.

1.5.3.1 con$objtbl Field

This field contains a pointer to the object array.

1.5.3.2 con$objnamtbl Field (Static or Dynamic, Depending on Implementation)

The name table field is a pointer to the table used to translate names for objects and logical names in this object container. The name table is described in Section 1.5.5, below.

1.5.3.3 condir$display_index Field (Static)

This field is only in the container directory.

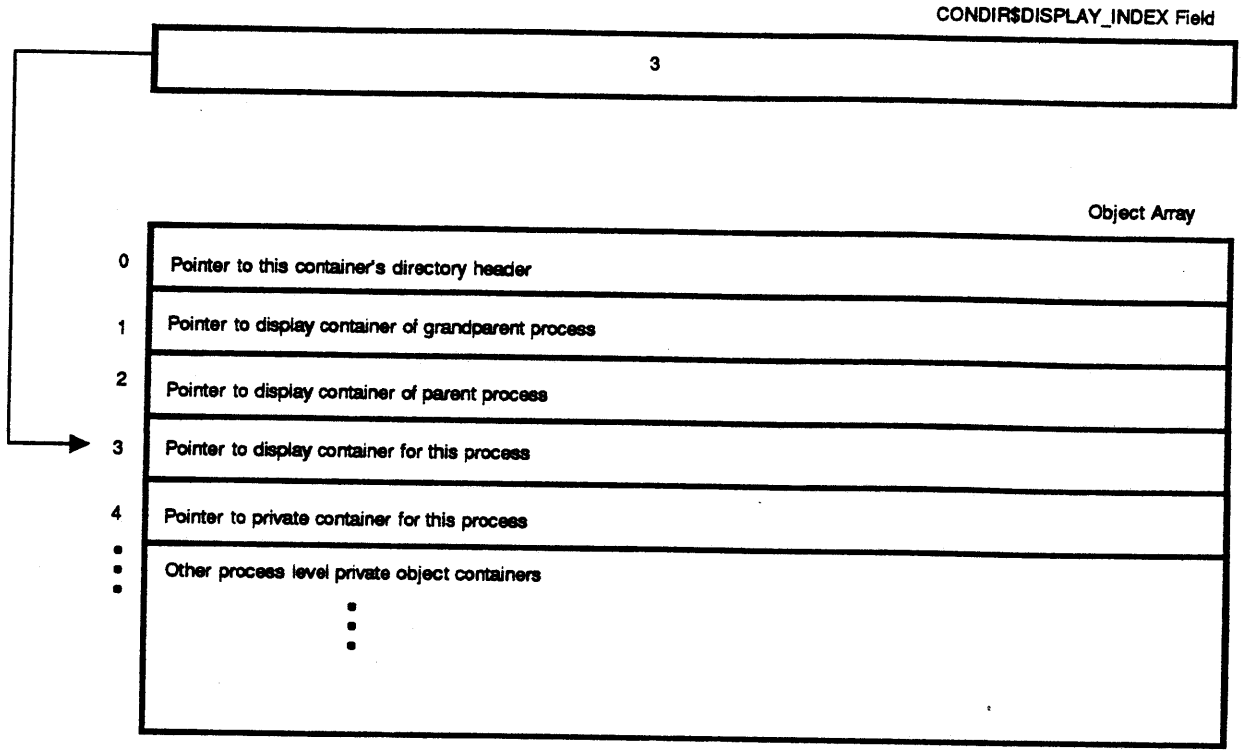
The purpose of the display index field supports the capability of a process to access objects in its ancestors' display object containers (as discussed in Section 1.2.6.5).

The first element in the object array (element 0) always points to its own object header.

In all cases except process level, the second element (element 1) always points to the first object container for that level. In these cases, the display index field contains the value one.

However, for process level container directories, the process's ancestors' display containers are pointed to by the second through nth elements (elements 1, 2, 3, etc.). In this case, the display index field contains the element number containing the current process's display object container. This element is then followed immediately by private process object containers. Figure 1–7 illustrates the use of the display index field in a process level container directory.

Figure 1–7: Container Directory Header Display Index Field Usage



1.5.3.4 objcon$mutex Field (Static)

The mutex field is a kernel mutex which is operated on by issuing calls to procedures within the kernel. These procedures are described in Chapter 6, The Kernel.

This mutex is used to synchronize access to certain fields in the object header.

1.5.4 Object Array Data Structure

The primary function of the object array is to hold a pointer to each created object in the object container. The format of the object array is shown in Figure 1–8.

The purpose of each of these fields is described below.

1.5.4.5 objtbl$max_index Field

This field indicates the current length of the object array by providing a count of the number of elements in it minus one. The value in this field is the maximum valid array index value.

This field value has an architectural limit of 1,048,575 (20 bits).

Figure 1–8: Container Object Array Format

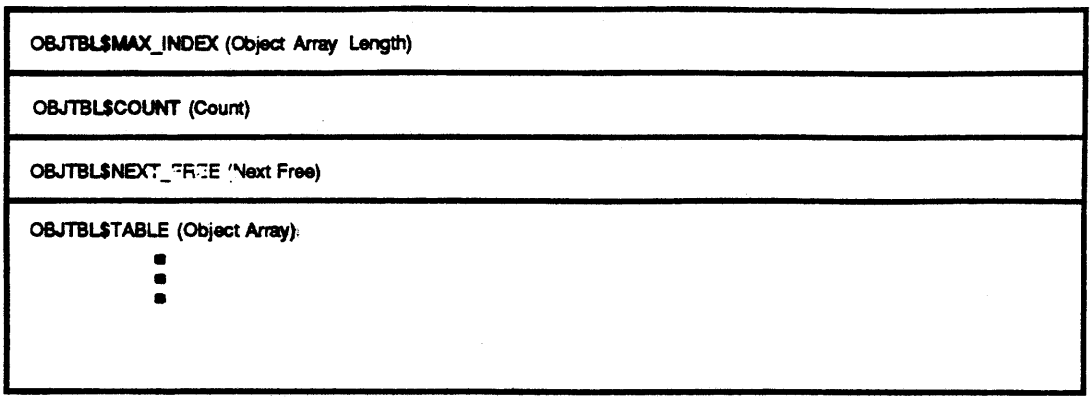


Figure 1–9: Address of Object Header

1.5.4.6 objtbl$count Field

This field is initialized to 1 when the object container is created, to represent the fact that this object container ID is in the array. It is then incremented each time an object ID within this object container is created. Similarly, it is decremented when an object ID in this object container is removed from the container as part of deletion.

1.5.4.7 objtbl$next_free Field

When created, an object container has a pointer to itself as the first element of the object array, and the remaining elements are free (unused). To facilitate quick allocation of free object array elements, they are organized into a linked "free list".

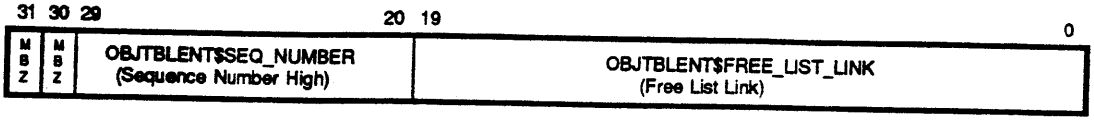
The next free field is the listhead of the free list, and contains the index of the next object array element to be allocated. The free list link field of the first element contains the index of the second, which contains the index of the third, and so on. The last element has a free list link value of zero to indicate the end of the list has been reached.

1.5.4.4 objtbl$table Field (Object Array)

Object array elements are used for two purposes. First, each object in the object container has an element in the object array that is associated with it. Such an element contains a pointer to the header of the object. In this case, the object element is referred to as "in use." The format of an object array element when it is in use is discussed below.

As discussed in Section 1.2.4, each time an object ID is allocated, it receives a new sequence number. This sequence number high field is associated with an object array element, and is stored within the element when the element is not in use. When the element is in use, the sequence number is moved to the sequence high field of the associated object's header. When the object is removed from the object container, the sequence number is incremented, reset to zero on overflow, and moved back into the object array element.

Figure 1–10: Free Object Array Element Format



The format of a free object array element is:

The free list link field is used to link all unused object array elements together on a single list for quick allocation (as discussed in the next section).

Notice that the most significant bit (MSB) of object array elements may be used to determine whether the element is currently in use or free. If the MSB is 1, then the element contains an address in system address space and the element is currently in use. If the MSB is 0, then the element is currently free and contains the next sequence number to use and a link to the next element on the free list.

Object array element index values begin at 0 and extend to the value in the *objtbl$max_index* field.

1.5.5 Name Table Data Structure

The purpose of the name table is to track both logical and object names within an object container.

When an object container is created, a name table must also be allocated, and the address of the name table must be stored in the name table field of the object container's body. The name table contains both object names which translate to an object ID and logical names which translate to an equivalence string.

The name table provides for a mechanism to translate names to name definition blocks. The actual implementation details of the name table need not be defined, only the functions provided by the name table are necessary. For example, the name table could be implemented as a hash table or a binary tree.

1.5.5.1 Name Definition Block

The name definition block contains information describing the name associated with a particular object or logical name. This information includes the type, mode, object ID, and other information about the name.

1.5.6 Object Naming Principles

Names are qualified by object type and mode. The combination of object type, mode, and name string is unique within a single object container.

For example, you can have two widget objects named SYS$ZEBRA in a single object container, but only if one is a user-mode object, and the other is a kernel-mode object. Likewise, you can have two user-mode objects named DEVICE_CONTROL in the same object container, but only if they have different object types.

Object names are limited to 255 characters in length. All object names are stored exactly as received (that is, case-sensitive) and may be looked up (translated) either case-sensitive or case-blind.

When a case-sensitive name translation is requested, either one or zero object IDs is returned. However, when a case-blind name translation is requested, many object IDs may match the name. To illustrate this, consider two objects, one named "pQr" and the other named "pqR". If a case-sensitive translation of "PQR" is requested, no object IDs is returned. If a case-blind translation of the same string, "PQR", is requested, then only the first name found which matches is returned.

Software which intends to use the case-blind feature of name translation is responsible for ensuring consistency. For example, all names strings could be converted to upper case before calling the object service routines. This prevents multiple names from matching in a case-blind search.

1.5.7 Naming

Names are used in the object architecture to label objects and equivalence strings. The same naming mechanism is used when a name is applied to either an object or an equivalence string. The only difference is the translation of an object name yields an object ID, while the translation of a logical name yields an equivalence string.

A name, like an object, exists within an object container. It is further qualified by the type of the object it names (logical names have the special object type of 0), and the mode in which it was created (user or kernel).

Through use of object name translation services a name can be searched for in any object container to which the user has access. The object name translation services provide recursion. See the description of *exec$translate_object_name* for more details.

The logical name translation services do not automatically provide recursion, rather it is the responsibility of the facility using the logical name translation services to provide for recursion, search lists, and parsing. This allows for a facility such as RMS to have different logical name evaluation rules.

1.5.7.1 Logical Names

Logical names are names that translate to equivalence strings. Each logical name can have up to 127 equivalence strings. The term "multivalued logical name" means a logical name with more than one equivalence string. Each equivalence string can be up to 255 bytes in length, and each equivalence string can have any, all, or none of the following attributes:

- Terminal—The equivalence string is not to be subjected to further logical name translations.
- Concealed—The application should not return the equivalence string to the user.

Logical names can have the following attributes:

- No show—The logical name should not be displayed in general show logical name services.
- No alias—The logical name cannot be duplicated in this table at an outer access mode. If another logical name with the same name already exists in the table at an outer access mode when a new logical name with no alias is created, it is deleted.
- Confine—The logical name is not copied from the process to its spawned subprocesses. This applies only to logical names at the process level in private object containers.

1.5.7.2 Object Services Providing Name Support

The following object services provide generalized name support.

- *exec$set_object_name*
- *exec$translate_object_name*
- *exec$clear_object_name*
- *exec$create_logical_name*
- *exec$translate_logical_name*
- *exec$delete_logical_name*

These routines are described in the Internal System Services Manual.

1.5.7.3 Container Directory Names

The three container directories to which the process control block points are named. The process container directory is named *exec$process_container_directory*, the job container directory is named *exec$job_container_directory*, and the system container directory is named *exec$system_container_directory*.

Each of the container directories has a name table (see Figure 1–2). This name table contains object container names which translate to the ID of an object container, and logical names, which translate to equivalence strings. When an object container is created, the name, if specified, is stored in the container directory's name table.

By creating "multivalued" logical names in the container directory name table, a search list for object containers can be created.

1.5.8 Object Name Translation

Object name translation involves taking a specified name and object type, and returning the related object ID (if any). The name table performs this function such that given a name, all name definition blocks containing that name are searched for the specified type.

The executive object support routine *obj$translate_object_name* provides the mechanism for object service routines to translate an object name to an ID.

```
PROCEDURE obj$translate_object_name (
    IN container_id : e$object_id;
    IN name : string (*);
    IN object_type : e$object_type;
    IN access_mode : k$processor_mode;
    IN case_blind : boolean;
    OUT object_id : e$object_id;
    ) RETURNS STATUS;
```

By specifying an OBJECT_TYPE of zero, the first object found which matches the name is returned.

obj$translate_object_name performs the following:

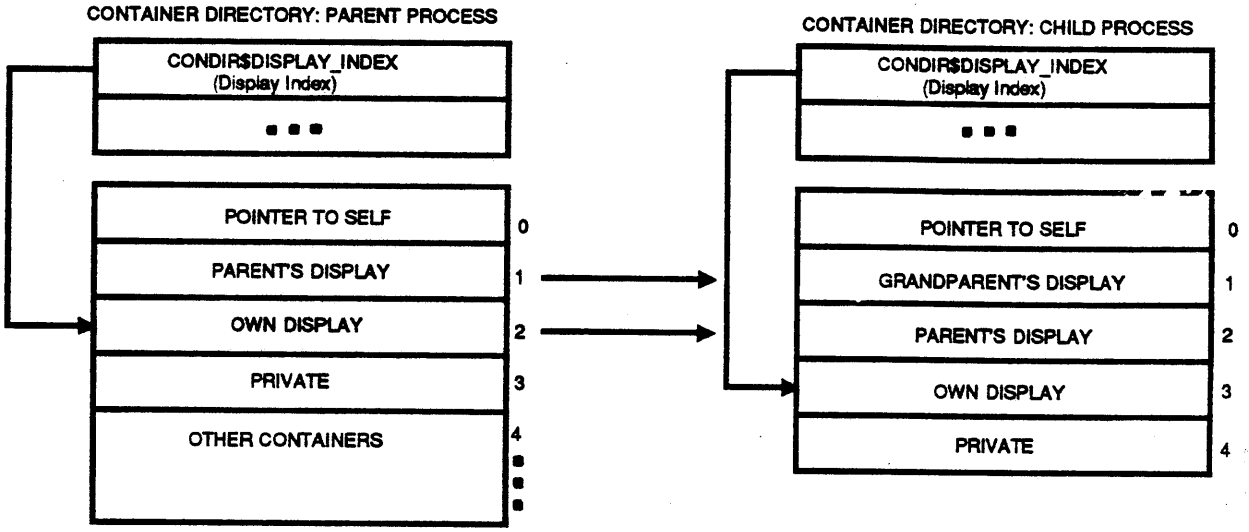
1. Acquires the directory-level mutex.
2. Translates the ID of the object container to a pointer to an object header and check access for read.
3. Ensures that the object header is a container or a directory.
4. Acquires the container mutex and releases the directory mutex if the object type is container.
5. Looks up the specified name in the container name table that matches the specified object type and mode.
6. Raises an error condition if the specified name is not found.
7. Returns the object ID which corresponds to the name if the name is found.
8. Releases the mutex.

1.5.9 Initializing a Process Level Container Directory

Container directories at system and job levels are always initialized to be empty (that is, they initially do not point to any object containers other than themselves). Container directories at the process level, however, must be initialized so that all of the process's ancestors' display containers are accessible. Also, in order to preserve the object ID of objects in its ancestors' display containers, it is necessary that they be pointed to by the same object array element in each process's container directory. To achieve this, process-display object containers are always the second through nth entries of a process level container directory.

A new process level container directory is initialized by copying elements starting at element one, and continuing through the value specified in the display index from the parent process's container directory. These entries are followed immediately by the new process's display and then private object containers. Figure 1–11 illustrates the initialization of a new (child) process's container directory.

Figure 1–11: Process Level Container Directory Initialization



In this example, the *condir$display_index* field of the parent's container directory indicates the last object container which must be copied to the new process's container directory. The new process's *condir$display_index* field is then updated to point to its own display container.

If the user deletes a display container, the table element within the container directory that pointed to the display container cannot be reused. If this were allowed, a private container could be copied as a display container.

1.5.10 System Global Variables and Data Structures

This section identifies the system global variables and data structures required to support the object architecture. The data structures that are accessible via known system locations include:

- System level container directory (see Section 1.2.8)
- System level directory mutex (see Section 1.2.6.2)
- Allocation mutex

1.5.10.1 OTD Objects

Each object of type OTD is created in a reserved system level object container. When a set of object services for an object type is loaded, the object service image is called at its initialization entry point. During the initialization operations, the corresponding OTD object must be created by calling the *obj$create_otd* service.

The *obj$create_otd* procedure has the following declaration:

```
PROCEDURE obj$create_otd (
    OUT otd_object_id : exec$object_id;
    IN  object_item_list : ITEM LIST CONFORM OPTIONAL;
    IN  transfer_state : boolean;
    IN  dispatcher_offset : integer;
    IN  shutdown_procedure : PROCEDURE e$shutdown_procedure;
    IN  remove_procedure : PROCEDURE obj$remove_procedure;
    IN  delete_procedure : PROCEDURE obj$delete_procedure;
    IN  allocate_procedure : PROCEDURE obj$allocate_procedure;
    IN  deallocate_procedure : PROCEDURE obj$deallocate_procedure;
    OUT otd_value : integer;
    );
```

At system shutdown, the operating system searches the OTD object container to locate each object type:

Each object type is unloaded by:

1. Setting the OTD's create disable flag (to prevent additional objects of that type from being created).
2. Calling the shutdown routine specified in that object type's OTD when all objects of a particular type have been deleted.

1.5.10.2 Allocation Mutex

The allocation mutex is a system-wide mutex which is acquired when an object's allocation is checked, modified, or deleted.

1.6 Relative Ordering of Object Architecture Mutexes

The following lists illustrates the relative ordering of object architecture mutexes. The mutexes are listed in order of lowest value to highest value.

Process level container directory mutex

- Process level object container mutex
- Job level container directory mutex
- Job level object container mutex
- System level container directory mutex
- System level object container mutex
- OTD type mutex
- Allocation mutex

1.7 Object Creation

1.7.1 Creating an Object

Each type of object is supplied with one or more service routines which create the object type. For example, a FRAMITZ object might be created with a *create_framitz* object service.

Object creation routines should have the following procedure declaration:

```
PROCEDURE exec$create_xxxxxx (          !where xxxxxx is the object name
  OUT xxxxxx_id : exec$object_id;
  IN object_independent_items : exec$object_parameters CONFORM OPTIONAL;
  IN object_dependent_items : exec$item_list CONFORM OPTIONAL;
    ) RETURNS STATUS;
```

The object independent items consist of the following:

- Object ID of object container in which the new object is to be created
- Name to associate with the object
- Access control information

The format and processing of object ownership and access control information is discussed in Chapter 11, Security and Privileges.

In addition to these standard parameters, object creation services may accept type-specific parameters. For example, an event creation service may accept an initial state of the event. The name of an object creation routine and the parameters allowed by that routine are specified by the designer of the object type.

1.7.2 Object Names

When creating an object, it is possible to assign a character string name to the object. This is particularly useful for objects that are to be shared. It allows one process to create an object (assigning it a name) and then, sometime later, another process can locate and access the object via its name.

Also, object names provide "create if" semantics. An attempt to create an object of same type, mode and name in the same container will fail, but the ID of the object which had the same name, type and mode is returned as is a warning status code.

1.7.3 Create Object Algorithm

The user calls an *exec$create_object* service specifying the container ID, ACL, returned object ID, and optionally a name. Other object type-specific arguments may be required.

Every create object service performs the following steps:.

1. Probes the user's arguments.
2. Calls *obj$create_initialize_object* passing in the following:
   - Object ID of OTD object
   - Pool type (paged or nonpaged)
   - The number of bytes of storage to allocate with the object header for the object body
   - The object name, if any
   - Ownership mode (kernel or user)
   - ACL
   - Transfer flag state

The *obj$create_initialize_object* routine locates the specified object type and allocates the storage.

At this point the object header has been created and initialized. The pointer count field is one, the object ID count and the container fields are zero.

3. Initializes the object body. This action is object type-dependent, for example, if the object supports waiting, a call to *k$initialize_xxxx*, where xxxx is a dispatcher initialization routine, is issued.

   At this point the object has been created but has no ID.

4. Attempts to add a pointer to the object header in the specified object container by calling *obj$insert_object_into_container* with the address of the object header, the container ID, the mode, and the object ID to be returned.

   obj$insert_object_into_container does the following:

   - Acquires the specified directory-level mutex.
   - Translates the ID of the object container to a pointer to an object header, and checks for write access. Ensures that the object header is a container.
   - Checks the transfer state of the container. If the container is transferable, and the object is not transferable, *obj$insert_object_into_container* returns an error status to the caller.
   - Acquires the container mutex.
   - Releases the directory mutex.
   - If an object name was specified, *obj$insert_object_into_container* checks to see if the same name already exists within this container with the same type and mode. If a name conflict occurs, *obj$insert_object_into_container* calls the OTD delete routine, releases the container mutex, and returns the ID that corresponds to the collided name.
   - The next free field in the object container's body is examined.
     - If the next free field is nonzero *obj$insert_object_into_container* does the following:
       * Removes the element specified in the next free field from the free list, and creates a new next free element.
       * Constructs the sequence number high field by using the value found in the object array, stored in the object header, and the address of the object header is stored in the element.
       * Constructs the object ID from the element number, the sequence number high, a random 10-bit number for the sequence number low, and the object index field of the object container.

         The object header is completed.
       * Sets the object ID count to 1, the container field points to the header of the object container, and so on.
       * Increments the create count field in the OTD for this object type using the RMALI instruction.
       * Increments the total ID field in the object container.
       * Releases the object container mutex, and returns the object ID to the user.
     - If the next free field is zero, *obj$insert_object_into_container* does the following:
       * Checks some resource quota to see if the object container can be expanded. If the object container cannot be expanded, *obj$insert_object_into_container* calls the OTD delete procedure. The container mutex is released, and a container full error is returned to the user.

- Allocates a new (larger) object array data structure.
- Copies the contents of the old object array to the new object array.
- Links all new object array entries beyond the length of the old object array into the free list.
- Sets the new object array's length.
- Changes the object array field in the object container to point to the new object array.
- Deallocates the old object array data structure.
- Creates the object ID as in the above case with a nonzero *objtbl$next_free* field.

At this point, if the object header is successfully added to the specified object container, both object ID count and pointer count fields contain the value one.

5. Adds the name, if any, to the object container name table. The container, the sequence number and object index fields are initialized.
6. Returns the object ID and status to the user.

If an ID cannot be created, the object type-specific delete procedure found in the OTD is invoked. Note that the container field and object ID count in the object's header were zero prior to the object's deletion. An error status is returned.

Once the object has been inserted in the specified container, the object creation routine may not reference the object body without calling the *obj$reference_object_by_id* routine.

The following status values could be returned from *obj$insert_object_into_container*:

- Invalid object container—Returned object ID is set to zero (error).
- Container full—Returned object ID is set to zero (error).
- Resource limit exceeded—Returned object ID is set to zero (error).
- Name collision—Returned object ID is set to the ID of the object with the same name and type (warning).
- Success—Returned object ID points to newly created object.

These rules cause all objects to be created in a uniform way. If two users attempt to create an object in the same container with the same name, type, and mode, only one is created, yet both users get returned a valid ID. One user gets a success code indicating the object was created, the other gets a warning code indicating the object already exists.

1.7.4 Object Modes

When creating an object, its mode (user or kernel) is established. Any object creation routine which is called from user mode creates a user-mode object.

Object creation routines which are called from kernel mode allow the mode to be specified as an argument. Object service routines that are called from user mode have the argument supplied by the jacket routine, which calls the corresponding entry point of the routine with the mode argument supplied.

This capability allows executive software to create objects that are either inaccessible to user mode or cannot be deleted from user mode.

Note that it is not possible to create an object owned by kernel mode in an object container owned by user mode. If this were allowed, the user would be able to delete the kernel mode object ID by deleting the container.

1.7.5 Object Access Protection

Object access protection is provided by a combination of user identifiers and access control lists (ACLs), and mode (user or kernel). This access protection information is stored for each object in its object header, and is processed by system-wide access validation routines without taking the object type into consideration.

Each object service routine is required to request access validation each time an object is accessed.

The exact format of the access protection information and procedures for performing access validation are provided in Chapter 11, Security and Privileges.

1.7.6 Object ID Translation

The executive procedure *obj$reference_object_by_id* provides translations from an object ID to a pointer to an object body. The procedure has the following declaration:

```
PROCEDURE obj$reference_object_by_id (
    IN object_id : e$object_id;
    IN object_type : e$object_type;
    IN access_mode : k$processor_mode;
    IN desired_access : e$access_type;
    OUT object_body : POINTER anytype CONFORM;
    ) RETURNS STATUS;
```

In order to translate an object ID to the address of the associated object, *obj$reference_object_by_id* performs the following steps:

1. Uses the level number as an index into the process control block to locate the corresponding level directory mutex.
2. Acquires the directory mutex.
3. Uses the level number as an index into the process control block to locate the corresponding container directory.
4. Compares the container directory index field to the size of the table to ensure that index is within the object array.
5. Uses the container directory index field to index into the object array.
6. Checks the resulting value to ensure that it is the address.
7. Uses the resulting system address to locate the object container header for the object container.
8. Compares the object index field to the size of the table to ensure that index is within the object array.
9. Uses the object index field to index into the object array.
10. Checks the resulting value to ensure that it is the address of an object.

    At this point, the address found in the object array is the address of the object header.
11. Compares both sequence number fields in the object header to the sequence number fields in the object ID. If they are not identical, that object ID is not valid.
12. Checks the object type to ensure this object is of the desired type.
13. Validates the intended access to the object by calling the security procedure *e$validate_access*.
14. Releases the directory mutex and returns the error status indicated by *e$validate_access* to the user if access is denied.
15. Does not validate allocation access if the intended access is show.

16. Checks to see if the object is allocated by examining the allocation information field in the object header.
17. If the allocation information field is not nil, then *obj$reference_object_by_id* performs the following steps:
    - Acquires the allocation mutex.
    - Checks the allocation information field in the object header again. If the field is nil, *obj$reference_object_by_id* releases the allocation mutex and continue as if the field was always nil.
    - Checks to ensure that the current thread is within the allocation class by calling the procedure *e$validate_allocation*.
    - Releases the allocation mutex.
    - Releases the directory mutex, and *obj$reference_object_by_id* returns the error indicated by *e$validate_allocation* to the user if allocation does not match.
18. Increments the pointer count field in the object header so the object cannot be deleted while a pointer to the object is held.
19. Releases the directory mutex, stores the pointer to the object body, and returns to the caller.

1.7.7 Object Deletion

The user calls the *exec$delete_object_id* service specifying the object ID of the object to delete.

The *exec$delete_object_id* routine performs the following steps:

1. Probes the user's argument.
2. Calls *obj$remove_obj_from_container* with the source ID and the mode of access.

The *obj$remove_obj_from_container* routine performs the following steps:

1. Acquires the source container directory-level mutex.
2. Acquires the source container mutex.
3. Translates the source object ID to a pointer.
4. Acquires the type-specific mutex.
5. If address of the source container matches the container field in the source object header, then *obj$remove_obj_from_container* zeros the container field and removes the object's name (if any) from the name table (in this case, this is the principal ID).
6. Decrements the object ID count of the source object.
7. Decrements the total ID field of the object container.
8. The *obj$remove_obj_from_container* routine performs the following steps if the resultant object ID count is zero:
    - Releases the type-specific mutex.
    - Checks the allocation block field in the object header. If the field is not nil *obj$remove_obj_from_container* does the following:
        — Acquires the allocation mutex.
        — Ensures the allocation block field is not nil.
        — Calls the type-specific deallocate routine, passing in the address of the object body and the allocation type.

  - — Clears the allocation pointer in the object's header.
  - — Unlinks the object allocation block and calls *obj$dereference_object*.
  - — Releases the allocation mutex.
  - — Returns the storage occupied by the object allocation block to the system.
- Removes the object ID and name from the source container. This frees the slot, updates the sequence number, and links it into the free list.
- Releases the source container directory level and the source container mutexes.
- Calls the type-specific remove object ID routine.
- Calls *obj$dereference_object* to decrement the *pointer count*.
- Returns to the caller.

9. If the resultant object ID count is one, the source object is temporary, and the container field in the source object's header is not zero, then *obj$remove_obj_from_container* performs the following steps:
   - Constructs the object ID of the principal object.
   - Deletes object ID and name from source container.
   - Releases the source container directory level, the source container, and the type-specific mutexes.
   - Calls an internal object routine to delete the principal ID. This routine is identical in function to *obj$remove_obj_from_container* with the following exception: The object ID count is examined once all the mutexes have been acquired. If the object ID count is not one, the principal ID is not deleted.
   - Returns to caller.
10. If the resultant object ID count is greater than one, or the object ID count is one and the source object is either not temporary or its principle object ID has already been deleted, then *obj$remove_obj_from_container* performs the following steps:
    - Removes the object ID and name from the source container.
    - Releases the source container directory level, the source container, and the type-specific mutexes.
    - Return to caller.

The procedure *obj$dereference_object* performs the following steps:

1. Decrements the pointer count field in the object header using the RMALI instruction.
2. If the pointer count field becomes zero, *obj$dereference_object* performs the following steps:
   - Issues a bug check if the object ID field is not zero.
   - Calls the OTD delete routine to deallocate the object body extensions and any associated structures.
   - Unlinks the object header from the OTD list.
   - Decrements the create count field in the OTD.
   - Deletes the object header and object body storage allocated in the *obj$create_initialize_object* object call.
3. Returns to the caller.

There is one interesting race condition that can arise during the deletion of an object ID that refers to a temporary object. This race condition, however, does not cause any system database to be corrupted. Assume that there are two threads that are simultaneously executing on two different processors: one is creating a reference to a temporary object, and the other is deleting a reference to the same temporary object. Further assume that the reference being deleted is the only reference to the object save its principle object ID.

Both threads execute the locking sequence using two different object IDs, and thus possibly acquiring different mutexes for the container directory and object container. The type-specific mutex, however, is the same, and one or the other of the threads acquires the mutex first. If the thread that acquires the mutex is the thread creating a reference, then nothing unusual can occur. If the thread that is deleting a reference acquires the mutex first, then the following anomaly can occur.

The thread deleting the reference discovers that the resultant object ID count is one, the subject object is temporary, and that its principle object ID has not been deleted. The reference object ID is deleted, the mutexes are released, and the Delete Object ID routine is called, specifying the principle object ID. But before the principle object ID can be deleted, the other thread completes the creation of the reference, thus incrementing the object ID count back to 2. It then releases its mutexes, whereupon the original thread now acquires the appropriate mutexes and checks the object ID count. The object ID count is not one and the principle object ID of the object is not deleted.

1.7.8 Transferring an Object Container

When a new process is created, it is given two object containers to serve as its display- and private-process level object containers. These object containers are created as part of process creation. See Chapter 5, Process Structure for more details on the Create Process and Job system service.
The object architecture provides support for the process and job creation services to transfer object containers from one job/process to a new job/process.

When an attempt is made to transfer an object container, every object in the container must meet the following conditions:

1. The thread issuing the system service must have write access to the object container.
2. The object being transferred must have an object ID count of one. This also prevents temporary objects from being transferred.
3. Reference objects must refer to a system level object.
4. The object being transferred must have the transfer action flag in the TRANSFER state.
5. If the object is allocated, the thread doing the transfer must pass the allocation check.

When an object is transferred, its name, if any, is transferred as well. If the object was allocated at the job, process, or thread level, then its allocation is transferred to the new entity being created. For example, if a job is being created, the object's allocation would transfer to the job level.

When a container is transferred, Mica performs the following steps:

1. Acquires the directory mutex.
2. Translates the container ID to an object header pointer with access as write.
3. Ensures that the resulting container object transferable. An error condition is raised if the container is not transferable. Displayed containers or the default private container are set as no transfer.
4. For each element in the object container Mica performs the following steps:
   - Checks to see if the object ID is a reference ID.
   - Checks the allocation of the object if the object ID is not a reference ID. If the allocation is at at thread level, then Mica transfers the allocation to the new entity.

If the object ID is a reference ID, checks to ensure that the principal ID is at the system level.

If all objects satisfy the criteria, removes the object container from the container directory and release the directory mutex.

5. Adds a pointer to the object container in the new container directory.
6. Updates the object header to point to the new container directory, also, the ownership of the objects needs to be changed to match the owner of the new entity.

1.7.9 Create Reference

When an object has a pointer to it outstanding, its data structures cannot be deleted. The executive can increment the pointer count field in the object header to prevent an object's storage from being deleted. A user-mode program may achieve similar behavior by creating a reference ID to the object ID.

Creating a reference ID creates another object ID for an object. When a reference ID is created for an object, the object ID count in the object's header is incremented. This allows a user to create a reference ID to an object, and thus ensures that the object's storage cannot be deleted until the reference object ID is deleted.

The level of the target container that receives the new reference ID must be less visible than the container of the principal object ID.

The *exec$create_reference_id* service, and the *exec$make_temporary* service both create reference IDs to the specified object.

The declaration for this executive function is:

```
PROCEDURE e$create_reference_id (
    IN mode : INTEGER;
    IN source_object_body : exec$object_id;
    IN target_container : exec$object_id;
    ) RETURNS reference_object_id;
```

The *e$create_reference_id* routine performs the following steps:

1. Raises an error condition if the level of the *target_container* is not less visible than the level of the source container.
2. Acquires the *target_container* directory-level mutex.
3. Acquires the *target_container* mutex.
4. Acquires the source container directory-level mutex.
5. Translates the source object ID to a pointer.
6. Checks for read access.
7. Checks the allocation class of source object container.
8. If the source object is of type container, then *exec$create_reference_id* releases the target container directory level, target container, and source container directory-level mutexes. An error condition is raised.
9. Allocates the object index in the target container.
10. If allocation of the object index fails, then *exec$create_reference_id* releases the target container directory level, target container, and source container directory-level mutexes. An error condition is raised.
11. Acquires the source object type-specific mutex.

12. Increments the object ID count of the source object.
13. Stores a pointer to the source object in target container at allocated index position.
14. Constructs the object ID of reference using the target container directory, the allocated target container index, and the sequence number of the source object.
15. Releases the target container directory level, target container, source container directory level, and type-specific mutexes.
16. Returns the object ID of newly created reference.

1.7.10 Make Temporary

Making an object temporary creates a new principal object ID for the object at a more visible level and marks the object as temporary. The original object ID becomes a reference object ID. If the object has a name, then the name is moved to the target object container.

When an object is made temporary, it is possible that a name conflict exists in the target container. The caller can specify that the object that is being made temporary is to be deleted and its object ID is to become a reference to the object which caused the name conflict. This provides the capability for two or more processes to share objects without regard to who created them first.

The declaration for this executive function is:

```
PROCEDURE e$make_temporary (
    IN mode : INTEGER;
    IN replacement : BOOLEAN;
    IN source : exec$object_id;
    IN target_container : exec$object_id;
    ) RETURNS reference_id : exec$object_id;
```

The *e$make_temporary* routine performs the following steps to make an object temporary:

1. If the level of the target container is not more visible than the source container, then *e$make_temporary* raises an error condition.
2. Acquires source container directory-level mutex.
3. Translates source object ID to a pointer.
4. Releases the source container directory-level mutex and raises an error condition if the source object is of type container.
5. Checks for delete access on source object.
6. Acquires target container directory-level mutex.
7. Acquires target container mutex.
8. Checks for temporary create access on target object container.
9. Acquires source object type-specific mutex.
10. If the source object ID is not the principle object ID, the source object ID count is not 1, or the source object is already temporary, then *e$make_temporary* releases the source container directory level, the target container directory, the target container, and the type-specific mutexes. An error condition is raised.
11. Checks for name conflict in target container if source object has a name.
12. If a name conflict exists and replacement is not specified, then *e$make_temporary* releases the source container directory level, the target container directory, the target container, and type-specific mutexes. An error condition is raised.

13. If a name conflict exists and replacement is specified, then *e$make_temporary* performs the following steps:
    - Raises an error condition if the object whose name conflicts is not temporary.
    - Increments the object ID count of the object in the target container whose name conflicts with the source object and save pointer to object.
    - Releases the target container directory level and target container mutexes.
    - Decrements the object ID count of the source object (the result is guaranteed to be zero).
    - Releases the type-specific mutex.
    - Calls the type-specific remove object ID routine for the source object.
    - Deletes the name of source object from source container.
    - Stores a pointer to the target container object in source container at the source object ID.
    - Retrieves the sequence number from the target object, and inserts in the source object ID.
    - Releases the source container directory-level mutex.
    - Returns the source object ID with the new sequence number inserted.
14. Allocates an object ID in the target container.
15. If allocation of the object index fails, then *e$make_temporary* releases the source container directory-level, the target container directory, the target container, and type-specific mutexes. An error condition is raised.
16. Stores a pointer to the source object in target container at the allocated index position.
17. Removes the source object name, if any, from its previous name table, and inserts it in the new target container name table.
18. Sets the principle object container address to the target container address.
19. Marks the source object temporary and indicates that a make temporary operation was performed.
20. Increments the object ID count of source object.
21. Releases the source container directory-level, the target container directory, the target container, and the type-specific mutexes.
22. Returns the source object ID.

1.7.11 Mark Temporary

The Mark Temporary routine allows an object to be marked as temporary after it has been created. This operation is used to cause a permanent object to be deleted when all its reference object IDs are deleted. If the object has no reference object IDs, then its principle object ID is deleted immediately. Objects can only be marked temporary that reside at the job or system levels.

The declaration for this executive procedure is:

```
PROCEDURE e$mark_temporary (
    IN mode : INTEGER;
    IN source : exec$object_id;
    )
```

The *e$mark_temporary* routine performs the following steps.

1. Raises an error condition if the level of the source object is a process.

2. Acquires the source container directory-level mutex.
3. Acquires the source container mutex.
4. Translates the source object ID to a pointer.
5. Checks for delete access on source object.
6. Acquires the source object type-specific mutex.
7. Sets the temporary flag in the source object and indicates that a mark temporary was performed. Note, if the object was already temporary, it does not change the state of the *objhdr$temporary_operation* flag.
8. If object ID count of source object is 1 and the source object ID is the principle object ID, then *e$mark_temporary* performs the following steps:
   - Decrements the object ID count (it is guaranteed to be zero).
   - Releases the type-specific mutex.
   - Calls the type-specific remove object ID routine.
   - Deletes the object ID and name from source container.
   - Releases the source container directory-level and source container mutexes.
   - Returns to the caller.
9. If object ID count of source object is not 1 or the source object ID is not the principle object ID, then *e$mark_temporary* releases the source container directory level, the source container, and the type-specific mutexes.
10. Returns to the caller.

1.8 Allocating an Object

An object may be allocated to one of the following classes:

- Identifier object
- User object (This is the user object which exists for each active user on the system.)
- Job object
- Process object
- Thread object

The identifier object class consists of all the identifiers in the rights database which have corresponding identifier allocation objects. Identifier allocation objects are created with the following service and exist in the system level container *mica$identifier_allocation*.

```
PROCEDURE exec$create_identifier_allocation (
    OUT identifier_allocation_id : exec$object_id;
    IN  object_item_list : exec$item_list;
    IN  identifier : INTEGER;
    ) RETURNS STATUS;
```

The active user ID class consists of the set of all user IDs currently active on the system. An object which is allocated to an active user ID is automatically deallocated when no users of that ID are currently on the system. For example, if an object is allocated to user KOLSEN, it is automatically deallocated when the last user with the ID KOLSEN is removed from the system.

The job class consists of the set of all threads within the job. When the job terminates, all objects allocated to the job class are automatically deallocated.

The process class consists of the set of threads within the specified process, and any thread within a descendant process. When the specified process terminates, all objects allocated to that process class are automatically deallocated.

The thread class consists of the thread that allocated the object. When the thread terminates, all objects allocated to the thread class are automatically deallocated.

The *e$allocate_object* argument that specifies the visibility of the object ID determines the allocation classes to which an object can be allocated. The rules for determining allocation class are as follows:

- If the ID is at the system level, the object may be allocated to any one of the five classes.
- If the ID is at the job level, the object may only be allocated to the job, process, or thread classes.
- If the ID is at the process level, the object may only be allocated to the process or thread classes.

1.8.1 Object Allocation Block

When an object is allocated the object header contains a pointer to the object allocation block. The object allocation block contains:

- A forward link to the next allocation block in this class.
- A backward link to the previous allocation block in this class.
- The allocation type (identifier, active user, job, process or thread).
- The allocation ID which identifies the allocation. For example, if the allocation is identifier, the ID of the identifier object is stored here.
- A pointer to the object header which refers to this allocation block.

Each allocation class has a listhead. The listhead for the thread allocation class is located in the tread object, the process in the process object, the job in the job object. The listhead for each active user is maintained in the "user" object structure. Job, process, thread and user objects are defined in Chapter 5, Process Structure. The listhead for an identifier object is located in the specific identifier object.

The listhead offset is stored in the OTD for the object type. This allows the object architecture to deallocate all object allocated to a particular object (like a thread) when that object is deleted.

There is a single mutex which guards access to all allocation class listheads. This mutex is known as the allocation mutex.

The *e$allocate_object* routine allocates the specified object to the specified allocation object. The declaration of this procedure is:

```
PROCEDURE e$allocate_object (
    IN object_id : exec$object_id;
    IN allocation_id  : exec$object_id;
    IN access_mode : integer;
    );
```

The *e$allocate_object* routine performs the following:

1. Translates the *allocation_id* with an access type of read to obtain the object class and other information.
2. Ensures that the calling thread is within the specified allocation class.
3. Ensures that the allocation class is compatible with the visibility of the specified object ID.

4. Translates the specified ID by calling *obj$reference_object_by_id*, with an access type of ALLOCATE.
5. Acquires an allocation mutex if the translation succeeds.

   This prevents other threads from attempting to allocate the same object simultaneously.
6. Checks to ensure that the object ID count for the object is 1.
7. Checks to ensure that the object is not currently allocated.
8. If all checks pass, *e$allocate_object* calls the OTD allocate routine for that object type, passing in the object body and the allocation type.
9. If the allocate routine does not return denial, *e$allocate_object* allocates an object allocation block, initializes the block, and links it into the appropriate allocation list.
10. Adds a pointer to the object allocation block to the object header and from the object header to the allocation block.
11. Releases the allocation mutex.
12. Dereferences the allocation object.
13. Dereferences the allocated object.
14. Returns status to the user.

\We need to be able to have a protected subsystem assume the allocation class as the thread whose behalf it is working \

1.9 Deallocating an Object

Allocated objects may be explicitly deallocated via a call to *exec$deallocate_object* or implicitly deallocated when an allocation class ceases to exist. For example, when a process terminates, any objects allocated to that process are implicitly deallocated.

The *e$deallocate_object* procedure deallocates the specified object. The declaration of this procedure is:

```
PROCEDURE e$deallocate_object (
    IN object_id : exec$object_id;
    ) RETURNS STATUS;
```

The *e$deallocate_object* routine performs the following:

1. Translates the specified object ID by calling *obj$reference_object_by_id* with an access type of DEALLOCATE.
2. If *obj$reference_object_by_id* succeeds and the object was allocated, then the thread is in the allocation class of the object.
3. Acquires the allocation mutex.
4. Checks to see if the object is currently allocated, if the object is not allocated, release the allocation mutex, call *obj$dereference_object* and return an error status of not allocated.
5. Calls the type-specific deallocate routine, passing in the address of the object body and the allocation type.
6. Clears the allocation pointer in the object's header.
7. Unlinks the object allocation block, and calls *obj$dereference_object*.
8. Releases the allocation mutex.
9. Returns the storage occupied by the object allocation block to the system.

10. Returns status to the user.

Implicit deallocation which occurs when an allocation class terminates. For example, upon thread termination, *e$deallocate_object* does the following:

1. Acquires the allocation mutex.
2. Removes the first entry from the terminating class's allocation list.
3. Uses the back pointer to the object header get the object type.
4. Calls the type-specific deallocate routine passing in the address of the object header and the allocation type.
5. Clears the allocation pointer in the object's header.
6. Unlinks the object allocation block.
7. Releases the allocation mutex.
8. Returns the storage occupied by the object allocation block to the system.
9. Repeats from step 1 until there are no more entries on the list.

1.10 Quotas and Objects

Objects are allocated from system paged and nonpaged pool, and as such are charged against the quota for a process or a job. The following rules indicate how and what is charged for object creation and reference. When an object is deleted, the same rules are used to return quota.

- Objects created in system level object containers are not charged against quotas. Objects at this level are permanent, and may outlive the life of the entity which created the object.
- Objects which are created at job or process level are charged to the level at which the object was created.
- If an object is permanent or was at one time permanent, (that is, marked as temporary) there are no charges for references. Note, objects of this type are in job or system containers.
- If an object was made temporary, a charge is made for the reference at the level of the reference. Since the creator of the object did not desire to create a permanent object, each referencer of the object is charged the total cost of the object. Note, the entity that initially created the object was charged for the object creation, and is not charged again when the object is made temporary.

\The exact methodology of charging and releasing quotas is still under discussion. One method is to add a field to the object header that indicates the charged amount for this object. This charged amount field would be used for creation, references, transfers, and deletion.\

\Chapter 5, Process Structure should describe a pair of procedures to charge and return quotas that allow the user to specify the amount, the type, and the entity to charge. \

1.11 Executive Support Functionality

This section describes the executive procedures which are invoked by object service routines for standard operations. Using a single set of executive procedures for common object operations enhances reliability and maintainability.

1.11.1 obj$reference_object_by_id

Given an object ID, type, mode and desired access, this procedure returns the address of the object body for the ID and increments the pointer count field for the object. An error is returned if the object ID is invalid.

```
PROCEDURE obj$reference_object_by_id (
    IN object_id : e$object_id;
    IN object_type : e$object_type;
    IN access_mode : k$processor_mode;
    IN desired_access : e$access_type;
    OUT object_body : POINTER anytype CONFORM;
    ) RETURNS STATUS;
```

1.11.2 obj$translate_object_name

Given an object container, object name, object type and access mode, this procedure returns the object ID corresponding to the name.

```
PROCEDURE obj$translate_object_name (
    IN container_id : e$object_id;
    IN name : string (*);
    IN object_type : e$object_type;
    IN access_mode : k$processor_mode;
    IN case_blind : boolean;
    OUT object_id : e$object_id;
    ) RETURNS STATUS;
```

1.11.3 obj$create_initialize_object

This routine is called with the number of bytes to allocate in addition to the object header. It allocates the storage from the appropriate pool, and initializes the object header.

```
PROCEDURE obj$create_initialize_object (
    IN otd_id : e$object_id;
    IN access_mode : k$processor_mode;
    IN acl : POINTER e$access_control_list;
    IN name : string (*);
    IN transfer : boolean;
    IN reference_inhibit : boolean;
    IN pool_type : e$pool_index;
    IN object_size : integer;
    OUT object_body : POINTER anytype CONFORM;
    ) RETURNS STATUS;
```

1.11.4 obj$insert_object_in_container

This routine accepts an object body pointer and a object container ID, and adds the object to the specified object container returning the new object ID.

```
PROCEDURE obj$insert_object_in_container (
    IN object_body : POINTER anytype CONFORM;
    IN access_mode : k$processor_mode;
    IN container_id : e$object_id;
    OUT object_id : e$object_id;
    ) RETURNS STATUS;
```

1.11.5 obj$insert_object_and_reference

This routine accepts an object body pointer and a object container ID, and adds the object to the specified object container returning the new object ID and pointer to the corresponding object body. The reference count on the object is increased by one.

This procedure is provided for object services which need to immediately reference an object after it has been created. It avoids race conditions with the object name colliding, the ID of the colliding object being returned, and before the object can be referenced, the object is deleted.

```
PROCEDURE obj$insert_object_and_reference (
    IN object_body : POINTER anytype CONFORM;
    IN access_mode : k$processor_mode;
    IN container_id : e$object_id;
    OUT object_id : e$object_id;
    OUT new_object_body : POINTER anytype CONFORM;
    ) RETURNS STATUS;
```

1.11.6 obj$remove_obj_from_container

This routine accepts an object ID and removes the object ID from its associated object container.

```
PROCEDURE obj$remove_obj_from_container (
    IN object_id : exec$object_id;
    IN access_mode :k$processor_mode;
    ) RETURNS STATUS;
```

1.11.7 obj$dereference_object

This routine decrements the pointer count of the specified object. If the resulting pointer count is zero, the object deletion routine for the specified object is invoked.

```
PROCEDURE obj$dereference_object (
    IN object_body : POINTER anytype CONFORM;
    );
```

1.11.8 obj$get_principal_object_id

This routine returns the principal object ID for a given object body. If the object has no principal ID, the invalid ID zero is returned.

```
PROCEDURE obj$get_principal_object_id (
    IN object_body : POINTER anytype CONFORM;
    ) RETURNS e$object_id;
```

1.11.9 obj$set_object_acl

This procedure updates the ACL field in the object header.

```
PROCEDURE obj$set_object_acl (
    IN object_body : pointer exec$object_body;
    IN acl : pointer exec$acl;
    ) RETURNS old_acl POINTER exec$acl;
```

1.12 Revision History, 31 AUG 1987

1. Removed ULTRIX fork and exec semantics.
2. Added Pillar record definitions.
3. Added e$insert_object_and_reference procedure.
4. Added allocation listhead offset to OTD.
5. Added quota rules.

1.13 Revision History, 04 MAY 1987

1. Added fork and transfer actions for reference IDs. The actions are essentially share.
2. Added identifier allocation objects.
3. Added *e$create_otd* service.

1.14 Revision History, 30 April, 1987

1. Changed names of container directories to *exec$level_container_directory*.
2. Changed *show_xxxx* routines to *get_xxx_information* routines.
3. Changed the Object type descriptor to be an object of type OTD.
4. Removed OTD flink from OTD body.
5. Changed all internal object routines which returned a pointer to an object header to return a pointer to the object body.
6. Added reference ID inhibit flag to object header.
7. Removed group owner and acl modification time from object header.
8. Added sequence number field to object header.
9. Added no show attribute to logical names.
10. Changed generic object creation procedure definition to reflect current thoughts on system services.
11. Changed semantics of when the OTD remove routine is called. It is now called after the object ID has been removed from the object container with no mutexes held.
12. Changed semantics of how a temporary object is deleted when it's object ID count is one so the race condition on deleting the principal ID is eliminated.
13. Added *e$set_object_acl* service.
14. Removed *exec$transfer_object* service.
15. changed the name of the following internal services to make their names more descriptive of their operations:
    1. *e$allocate_initialize_object* becomes *e$create_initialize_object*
    2. *e$deallocate_object* becomes *e$delete_object*
    3. *e$create_object_id* becomes *e$insert_object_into_container*
    4. *e$remove_object_id* becomes *e$remove_object_from_container*

5. *e$translate_id* becomes *e$reference_object_by_id*
6. *e$decrement_pointer_count* becomes *e$dereference_object*

1.15 Revision History, 20 March, 1987

1. Removed alias object.
2. Removed allocate object.
3. Added reference IDs.
4. Added allocation classes.
5. Added allocation mutex.
6. Added OTD object flink/blink and sequence number.
7. Added exec action to object header.
8. Added allocate procedure to OTD.
9. Added deallocate procedure to OTD.
10. Removed transfer procedure from OTD.
11. Added fork action of transfer.
12. Added ACL time to object header.
13. Added algorithms for the following:
    1. create object
    2. delete object
    3. remove object
    4. fork action
    5. make temporary
    6. mark temporary
    7. translate ID
    8. allocate
    9. deallocate
    10. translate name

1.16 Revision History, 28 January, 1987

1. Restructure and reorganize material.
2. Remove CIT and replace with container directory
3. Added additional allocation size field to OTD.
4. Added container pointer to object header.
5. Added object creation rules.
6. Added object deletion rules.

1.17 Revision History, 14 January, 1987

1. Eliminated concept of nested object containers. Object containers are in a flat name space.
2. Eliminated concept of support for general wait mechanism in objects. Wait is supported by using a kernel support object within the object.
3. Simplified naming and eliminate CINB, ENB, and other naming structures.
4. Added support for logical names.
5. Simplified locking rules on containers and objects.
6. Added rules for alias object creation.
7. Added allocate objects.
8. Added rules for temporary objects.
9. Added FORK dispatch element to SSVT for support of ULTRIX style fork services.
10. Changed security mechanism to ACL based and removed object types reserved for security.
11. Changed definition of reference count in object header.
12. Removed alias count and added share count to object header.
13. Removed container lists. This functionality may be obtained through logical names.
14. Added definitions of executive routines for internal object support.
15. Added definitions of object services for user object support.

The following system services will exist in the chapter on MICA system services. They have been included with the object chapter for the chapter review and have not been reviewed or finalized.

Please note that the names for various parameters may not match the current naming guidelines. These will be remedied in the future.

```
!----------------------------------------------------------------------------
! Template CREATE_object and GET_object_INFO procedure definitions.
!----------------------------------------------------------------------------

PROCEDURE exec$create_xxxxxx (
    OUT principal_id : exec$type_object_id;
    IN obj_independent_items : exec$type_item_list (*) CONFORM OPTIONAL;
    IN obj_dependent_items : exec$type_item_list (*) CONFORM OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_xxxxxx_information (
    IN object_id : exec$type_object_id;
    IN items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;


!----------------------------------------------------------------------------
! General object service routines.
!----------------------------------------------------------------------------

PROCEDURE exec$translate_object_name (
    IN object_name : STRING (*);
    IN object_type : LONGWORD;
    IN container_id : exec$type_object_id OPTIONAL;
    IN container_name : STRING (*) OPTIONAL;
    IN case_blind : BOOLEAN = FALSE;
    IN substitute : BOOLEAN = FALSE;
    OUT principal_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_object_information (
    IN object_id : exec$type_object_id;
    IN object_id_items : exec$type_item_list (*) CONFORM OPTIONAL;
    IN object_items : exec$type_item_list (*) CONFORM OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$allocate_object (
    IN object_id : exec$type_object_id;
    IN allocation_object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_allocated_object_id (
    IN allocation_object_id : exec$type_object_id;
    IN object_type : LONGWORD OPTIONAL;
    IN OUT context : QUADWORD;
    OUT object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;
```

```
PROCEDURE exec$deallocate_object (
    IN object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$set_object_name (
    IN principal_id : exec$type_object_id;
    IN object_name : STRING (*);
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$clear_object_name (
    IN principal_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$delete_object_id (
    IN object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$create_reference_id (
    IN principal_id : exec$type_object_id;
    IN object_container_id : exec$type_object_id OPTIONAL;
    OUT reference_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$transfer_obj_make_temp (
    IN principal_id : exec$type_object_id;
    IN object_container_id : exec$type_object_id;
    IN replace : BOOLEAN = FALSE;
    OUT reference_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$make_object_temporary (
    IN object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;



!----------------------------------------------------------------------
! Container directory object service routines.
!----------------------------------------------------------------------

PROCEDURE exec$get_contr_dir_information (
    IN container_directory_id : exec$type_object_id;
    IN items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_object_container_id (
    IN container_directory_id : exec$type_object_id;
    IN OUT context : QUADWORD;
    OUT object_container_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;
```

```
!----------------------------------------------------------------------------
! Object container object service routines.
!----------------------------------------------------------------------------

PROCEDURE exec$create_object_container (
    OUT principal_id : exec$type_object_id;
    IN obj_independent_items : exec$type_item_list (*) CONFORM OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_obj_contr_information (
    IN object_container_id : exec$type_object_id;
    IN items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_object_id (
    IN object_container_id : exec$type_object_id;
    IN object_type : LONGWORD OPTIONAL;
    IN OUT context : QUADWORD;
    OUT object_id : exec$type_object_id;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;




!----------------------------------------------------------------------------
! Object type descriptor object service routine.
!----------------------------------------------------------------------------

PROCEDURE exec$get_otd_information (
    IN object_type_desc_id : exec$type_object_id;
    IN items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;




!----------------------------------------------------------------------------
! Logical name object service routines.
!----------------------------------------------------------------------------

PROCEDURE exec$create_logical_name (
    IN logical_name : STRING (*);
    IN container_id : exec$type_object_id OPTIONAL;
    IN container_name : STRING (*) OPTIONAL;
    IN create_if : BOOLEAN = FALSE;
    IN logical_name_items : exec$type_item_list (*) CONFORM;
    IN equivalence_name_items : exec$type_item_list (*) CONFORM;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$translate_logical_name (
    IN logical_name : STRING (*);
    IN container_id : exec$type_object_id OPTIONAL;
    IN container_name : STRING (*) OPTIONAL;
    IN case_blind : BOOLEAN = FALSE;
    IN logical_name_items : exec$type_item_list (*) CONFORM OPTIONAL;
    IN equivalence_name_items : exec$type_item_list (*) CONFORM OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;
```

```
PROCEDURE exec$delete_logical_name (
    IN logical_name : STRING (*) OPTIONAL;
    IN container_id : exec$type_object_id OPTIONAL;
    IN container_name : STRING (*) OPTIONAL;
    OUT deleted_container_id : exec$type_object_id OPTIONAL;
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;

PROCEDURE exec$get_logical_name (
    IN container_id : exec$type_object_id;
    IN OUT context : QUADWORD;
    OUT logical_name : STRING (*);
    ) RETURNS INTEGER; !STATUS;
    EXTERNAL;
```