The Alpha Operating System: Kernel Interface

J. Duane Northcutt

TR #88061

Archons Project Technical Report

June 30, 1988

The Alpha Operating System: Kernel Interface

J. Duane Northcutt

June 30, 1988

Archons Project Technical Report #88061

This research was supported in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC or the U.S. Government.

Table of Contents

Abstract **1**

1 Introduction **2**

2 Kernel Entry Points **3**

2.1 Traps 3
2.1.1 Operation Invocation 3
2.1.1.1 Client Invocation 5
2.1.1.2 Kernel Invocation 5
2.1.2 Exception Blocks 6
2.1.3 Event Logging 7
2.2 Interrupts 7
2.2.1 On-Board DUSART 8
2.2.2 On-Board Timer 8
2.2.3 Inter-Processor Communications 8
2.2.4 Spurious Interrupts 8

3 System Service Objects **9**

3.1 Manager Objects 9
3.1.1 Object Manager 9
3.1.2 Thread Manager 11
3.1.3 Semaphore Manager 12
3.1.4 Lock Manager 13
3.2 Support Objects 14
3.2.1 Semaphore 14
3.2.2 Lock 15
3.2.3 Time Services 17
3.3 I/O Objects 18
3.3.1 TTY 18
3.3.2 Network I/O 19
3.4 Miscellaneous Object 20

References **22**

Appendix I: Kernel Interface Usage Examples **23**

List of Figures

Figure 1 Invocation Parameter Block Formats 4
Figure 2 Object Manager Object 10
Figure 3 Thread Manager Object 11
Figure 4 Semaphore Manager Object 12
Figure 5 Lock Manager Object 13
Figure 6 Semaphore Object 14
Figure 7 Lock Object 16
Figure 8 Time Services Object 17
Figure 9 TTY Object 18
Figure 10 Network I/O Object 19
Figure 11 Temporary Services Object 20

List of Figures

Table 1 Lock Compatibility Table.. 16

Abstract

This report describes the programmer's interface to the Release 1 kernel of Alpha. Included in this description is an enumeration of all of the points of entry into the kernel (i.e., software generated traps and hardware generated interrupts). Also described in this document are the units of kernel functionality provided to the kernel's clients in the form of objects (known as *system service objects*).

In the design of the Alpha kernel an attempt was made to make the kernel interface be simple and uniform. This document provides an illustration that these goals have been accomplished. There are a minimum number of entry points into the kernel, and most services are provided by a small number of objects (each of which has an equally small number of operations).

1 Introduction

This document describes the functional interface provided by the Release 1 kernel of Alpha. A major objective in the functional design of the kernel was to provide all system services in a simple and uniform manner. This led to the decision to provide all kernel functions in such a way as to appear to the client as though they are provided by a collection of kernel-defined objects. The functions that are traditionally accessed via traps in more typical operating systems (commonly, one trap for each service), are accessed in Alpha by the single trap that performs the operation invocation function.

A simplified description of the Alpha kernel interface includes only the operation invocation mechanism, but a more complete description includes the definition of all of the services provided by the kernel in the form of objects, and all of the other points of kernel entry.

In this report an enumeration of the entry points into the kernel is given, followed by a list of the services provided by the kernel in the form of objects. The kernel entry mechanisms fall into two categories—traps and interrupts—each of which provide a separate means of entry into the kernel. The kernel services that are provided in the form of objects, known as *system service objects* are accessed through the operation invocation trap.

The bulk of the kernel's interface is involved in providing support for the basic abstractions—i.e., objects, operation invocation, and threads. The operation invocation mechanism is provided to the client as a programming language construct [Shipman 88], and the kernel support for objects and threads is provided by system service objects.

This report represents the actual implemented interface for Release 1 of the Alpha kernel. A description of the intended functional behavior of the Alpha operating system is provided in [Northcutt 88a], while [Northcutt 88b] contains a description of the programming model that the Alpha kernel is intended to support, and [Shipman 88] gives a description of the programming interface that is provided on top of the kernel interface described in this document.

In the following chapters the kernel entry points (traps and interrupts) and system service objects are defined. In an appendix, an example of the manner in which the Alpha object-language preprocessor makes use of the kernel interface mechanisms is presented.

2 Kernel Entry Points

The only means of entering the Alpha kernel is either by software trap or hardware interrupt. Each of these entry points is described in this document—first the system traps are discussed, followed by a discussion of the system interrupts.

The primary entry point into the kernel (and abstractly, the only one) is via the operation invocation facility. Fundamentally, the Alpha kernel interface consists of the object invocation mechanism and a collection of kernel-defined objects. The operation invocation mechanism is made available by the kernel as the only *system call* (implemented by way of a trap instruction), while the kernel-provided objects are kernel routines that present the client with an object interface (and, as part of the kernel's code, are available as soon as the kernel starts up on a node).

In Release 1 of Alpha, all machine exceptions (e.g., divide by zero and unimplemented instructions) are handled by the monitor, and so are not described as part of the kernel interface.

2.1 Traps

In addition to the primary entry point provided by the operation invocation facility, two other trap entry points are provided in Release 1. These are for exception block management and event logging, and both are intended to be accessed through the operation invocation facility, but were implemented as separate traps for development purposes and will integrated with the operation invocation facility in a future release.

2.1.1 Operation Invocation

In Alpha, operation invocations are similar to traditional remote procedure calls (RPCs), there is an entry into the kernel and then a return from the kernel to the invoker when the operation is complete. Each thread has a collection of parameters that are brought into a target object by a thread on the invocation of an operation and a set that are returned from the object when the operation completes. Collectively, these parameters are known as the "incoming" parameters of a thread within an object. In addition, a thread that invokes an operation on an object uses a collection of parameters that contain the values sent to, and subsequently returned from, a target object. These parameters are known as "outgoing" parameters. The parameters that pass into an object on an operation invocation are known as *request* parameters, and those that are passed back from the target object when the invocation is complete are known as *reply* parameters. The detailed format of both the parameter blocks used in Release 1 of Alpha is shown in Figure 1, with the request parameters shown in part 1a, and reply parameters in part 1b.

All parameters are packed into the contiguous memory following the parameter block header. The header portion of a parameter block contains all of the control information needed from the client for an invocation, and the parameter portion contains both simple parameters (i.e., arbitrary C data types), and capabilities (i.e., indexes into the invoking object's c-list). The invoker passes capabilities in the form of indexes into the invoking object's c-list. At the beginning of an invocation the kernel validates all passed capability indexes and translates them into object descriptors. Should any of these tests fail, the invocation is not performed and an error indication is returned in the reply parameter block

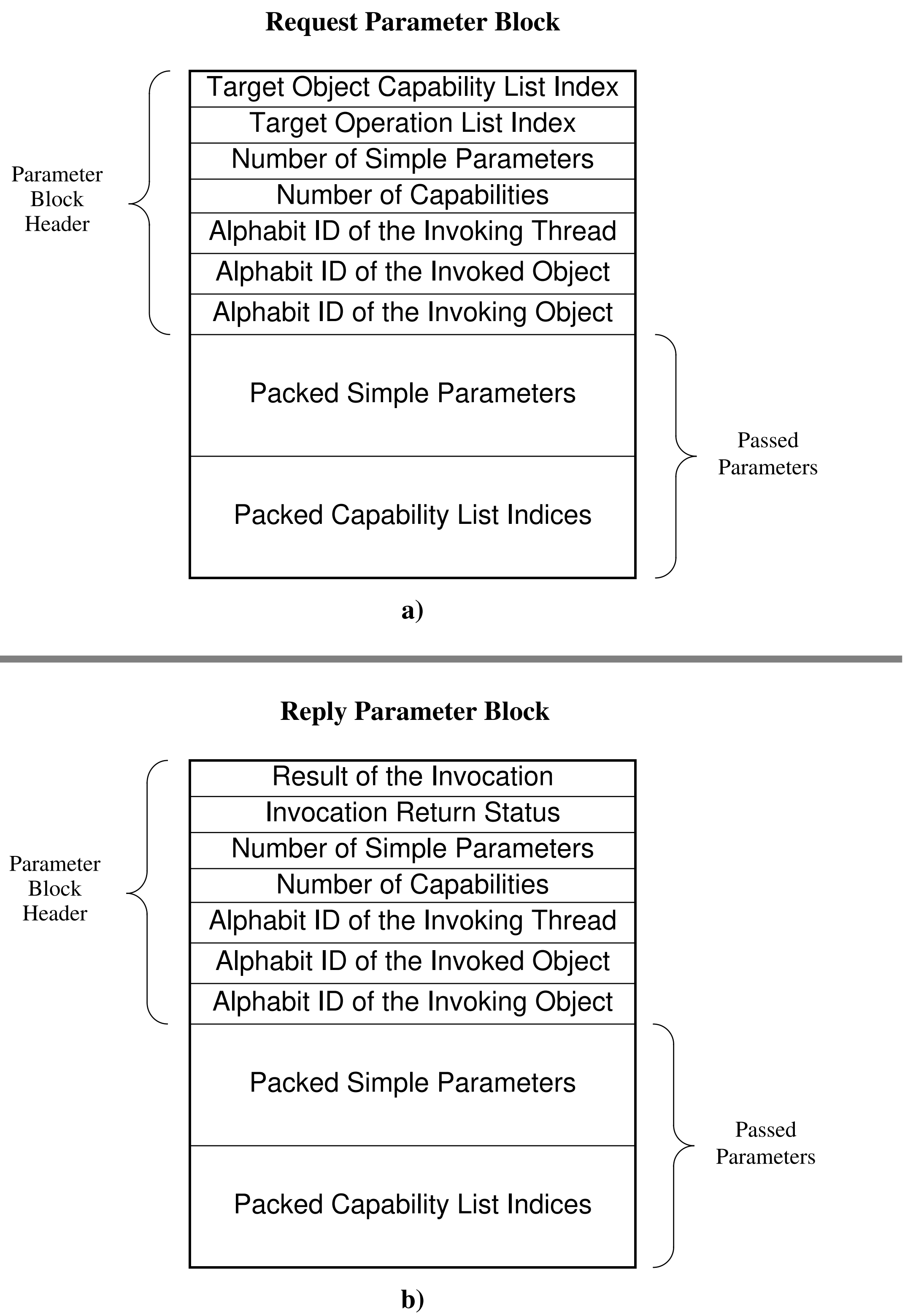

Figure 1: Invocation Parameter Block Formats

When an invoked operation is complete, capability indexes that are to be returned as reply parameters to the invoker are similarly validated and translated by the kernel, and any error indications are likewise returned with the invocation†.

The logical presentation of the operation invocation facility to the client involves the use of a set traps, and a collection of parameter passing conventions. There are two different system locations from which operation invocations can be performed—i.e., from client objects, and from kernel objects. While operation invocations from each of these points is made to appear the same to the user, for reasons of simplicity and efficiency, a different trap entry point is used for each of these types of invocations. The following discussion describes the behavior of the operation invocation facility in each of these cases.

2.1.1.1 Client Invocation

The most common form of operation invocation emanates from client objects and is known as *client invocation*. This type of invocation involves the movement of parameters between object address spaces. To accomplish this efficiently, the invocation parameters are placed in separate pages so that the memory management hardware can be used to map the parameters among objects (as opposed to copying)†.

To perform a client invocation, the appropriate values are loaded into the request part of the outgoing parameter block page, and a trap instruction (i.e., `trap #10`) is executed. On client invocations, the location of the parameter page is known implicitly by the kernel, which accesses the appropriate parameter pages as it carries out invocation.

In a fashion very similar to that of client invocations, the kernel must be entered upon the completion of an operation invoked on a client object. This requires that the target (client) object marshal return parameters and execute a trap instruction (i.e., `trap #11`) to return to the kernel on its way back to the invoking object. An activity very similar to that performed on the request side of the invocation is carried out as a part of this trap.

When the invoked operation is complete, return values are placed into the reply side of the invoking thread's outgoing parameter block, any passed capabilities are validated, inserted into the invoking object's c-list, and then translated into c-list indices. When this has been done, the operation is allowed to complete and the invoking thread resumes execution within the object from which the invocation was made.

2.1.1.2 Kernel Invocation

The invocation of operation on objects from within a kernel object is performed via a *kernel invocation*. From the client programmer's point of view this form of invocation provides a function identical to that of client invocations, except in that the parameter blocks do not exist in a well-known location in virtual address space, and the invoking object is already within the kernel's address space. Thus, one difference between this form of invocation and client invocation is that pointers to the invoking thread's outgoing, request and reply parameter blocks are pushed onto the kernel stack prior to the execution of the invocation trap (i.e., `trap #9`).

The only other difference is that there is no need to trap back to the kernel when an operation invoked on a kernel object completes. In such a case, the invoking object is already in the ker-

†*This is an area where the functionality has been modified in Release 2.*

nel's address space, so an RTE from the invoking trap is all that is needed to return from an invocation on a kernel object.

2.1.2 Exception Blocks

The Alpha programming model includes a notion of *exception blocks* which are defined as contiguous regions of code within an object that have special regions of code (known as *exception handlers)* associated with them, that are executed when exceptions occur while threads are executing within the exception blocks. The kernel provides mechanisms to permit the programmer to define the beginning and end of exception blocks and to define the associated exception handlers for these blocks.

While these exception block mechanisms would best be provided by invocations on the thread executing within an exception block, the current implementation provides these functions by an independent trap entry point. The exception block trap (i.e., trap #12) provides the mechanisms that define the start and end of an exception block, as well as the beginning and end of each block's exception handler.

The command types implemented in Release 1 are: begin/end a generic exception block, begin/end a time constraint block, begin the system-defined exception handler, and end an exception handler. The exception block begin and end commands can be executed at any point within an object, the intent is that all begin commands have matching end commands within the same object. This, however, is not a requirement—if more begin than end operations are executed within an object, the kernel automatically (effectively) performs the necessary number of end operations when an operation completes. In addition, exception blocks (of all varieties) can be nested within an object.

At the point in an object where an exception block command is to be performed, an identifier for the type of operation to be performed is pushed onto the stack (along with any additional parameters needed by the given operation) and the trap instruction is executed. The same trap instruction is used to perform exception block operations from both kernel and client objects.

When a begin operation is executed, the kernel saves the state of the executing thread (e.g., stack frame pointer and processor register set) and returns a false condition. When an end command is executed, the most recently saved thread state is discarded. Exception blocks manage thread state in a stack-oriented fashion, this is to say that begin operations pushes the thread's state and end operations pop the saved thread state. Should an exception occur while the thread is executing within an exception block, the kernel returns the thread to each of the previously saved locations in turn, restores the thread's state to what it was when it executed each block's begin command, and returns a true condition. With these mechanisms the kernel's clients can define a variety of different exception blocks and customized exception handling code.

The two types of exception handling blocks defined in Release 1 are known as *generic* and *time constraint* exception blocks. Generic exception blocks allow exception handling code to be associated with arbitrary regions of code within an object. Time constraint exception blocks are used to define time constraints on threads, and unlike all of the other commands which only take the command type parameter, time constraint begin commands include thread time constraint parameters. Time constraint parameters are specific to the scheduling algorithm used and are pushed onto the stack following the exception block type code. The exact parameters required for time constraint blocks in Release 1 are defined in [Shipman 88].

2.1.3 Event Logging

An event logging facility was added to Alpha for both system and application program performance monitoring. In Alpha, events are programmer-defined points in a program's execution stream. The kernel logs all (enabled) events in a buffer area that is set aside for this purpose within the kernel's address space.

The kernel's debugging facility permits the interactive examination of event logs, and the event logs can also be uploaded to a development host (via the monitor-provided TFTP facility) and examined with a post-processing tool (described in [Northcutt 88d]).

The kernel's event logging mechanism requires that a parameter indicating the event type (that is used as a index into a event print-string table), and an arbitrary user-defined parameter (that is used to further disambiguate the event). These parameters are pushed onto the stack and then the event trap is executed (i.e., trap #14), following which, the kernel adds a timestamp and logs the event into the kernel's event buffer.

This trap operation can be used from within both kernel and client objects, and it returns promptly following the timestamping and logging of the event.

2.2 Interrupts

In addition to traps, interrupts are used to gain entry to the kernel. The application processor (the processing element within a node on which the kernel proper executes) can receive interrupts from a variety of sources, either from on-board devices or from devices on the node's Multibus.

The general philosophy for the handling of interrupts in Alpha involves their conversion into activities performed by threads, in as prompt a fashion as is possible. This suggests that very few instructions are executed under interrupt—typically all that is performed by an interrupt handler is the initiation of a thread. The reason for this approach is based on the belief that the scheduler should manage all application processing cycles, and traditional interrupt processing subverts this process by asynchronously diverting processing cycles to activities (under the direction of the interrupt handling hardware and not the operating system's scheduler). By converting the activities typically associated with interrupt handlers into a form that the kernel's scheduler can manage, allows all processing cycles according to a globally consistent, time-driven application processor cycle management policy.

The application processor recognizes seven levels of hardware-vectored interrupts, which (in the Alpha testbed) are assigned as follows:

- **Level 7:** Non-Maskable Interrupt (this interrupt is handled by the monitor)
- **Level 6:** On-Board Dual USART
- **Level 5:** On-Board Timer
- **Level 4:** Inter-processor Communications
- **Levels 3-1:** Multibus Interrupts

Each of the kernel's interrupt handling routines and their sources are described in the following discussion.

2.2.1 On-Board DUSART

The on-board DUSART is used to provide basic serial communications for each node. The application processors use the A port as the node's console connection, and the B port is used to communicate with the other processors in a node. The monitor uses port A as its (default) console line, and (once loaded and running) the kernel also uses this port for console communications.

The kernel provides a ***TTY*** object that encapsulates the serial communications unit and provides basic I/O functions to the Alpha system and its applications. The ***TTY*** object takes characters in as parameters and sends them across the serial line, and also accepts characters from the serial line and unblocks threads that are waiting for incoming characters. In addition, the object provides functionality equivalent to the UNIX raw and cooked I/O modes.

2.2.2 On-Board Timer

The application processor uses its on-board, programmable, counter/timer unit to generate a constantly running, "real-time" system clock for the kernel. Because the timer runs constantly and is of finite length, the counter may "roll-over" during the lifetime of the system. Therefore, the timer hardware is configured to generate interrupts on counter overflows, so that the interrupt handler can increment the software portion of the system clock value when an overflow occurs.

2.2.3 Inter-Processor Communications

The processors within an Alpha testbed node exchange messages via shared memory, and interrupts are used to notify the destination processor of the arrival of a message. Such messages are sent along uni-directional logical links (known as *channels*) among the processors in a node. Inter-processor messages are buffered within channels until they are removed by the destination subsystem.

The kernel executing on the application processor receives interrupts from the various kernel subsystem processors to indicate message arrivals. The application processor's inter-processor communication interrupt handler polls all of the incoming channels, removing and acting on each message it finds, until all channels are empty. The only action that is typically carried out in response to an incoming message is the unblocking of a thread.

2.2.4 Spurious Interrupts

All other interrupts are not used by the Release 1 kernel, and therefore the kernel always runs at minimum priority of three, so as to block out all other interrupts (i.e., Multibus interrupts such as Ethernet controllers, disk controllers, and interrupts directed from the applications processor to the co-processors).

Just to be sure, the kernel installs a spurious interrupt handler for all unused interrupts. This handler does nothing but print a console message and return.

3 System Service Objects

The bulk of the kernel's functionality is provided through services accessed through the operation invocation facility. These services are organized into groups and made to appear as objects, known as "system service objects." These pseudo-objects provide support for the Alpha basic abstractions, as well as a range of supporting facilities.

The following sections describe the interface and function of each of the system service objects that the kernel provides in support of the system's programming model.

3.1 Manager Objects

The object programming paradigm used in Alpha suggests that all system activities be performed by the invocation of operations. For example, all manipulation of objects should be done by invoking operations on the object itself. However, there are a class of operations that do not lend themselves well to this uniform, invocation-based approach—e.g., it is not possible to invoke an operation on an object that does not yet exist, and so this approach cannot be used for object creation. For this reason, the Alpha kernel provides a collection of special *manager* objects to carry out the operations that are not naturally performed by invoking operations on an entity directly.

Once an entity is created, it is possible to perform (almost) all subsequent operations on it directly. Therefore, while it was not strictly necessary to do so, the DELETE operation has been included with the CREATE operations provided by manager objects (for reasons of symmetry).

The manager system service objects for objects, threads, semaphores, and locks are described in the following subsections.

3.1.1 Object Manager

In support of the Alpha object abstraction, the kernel provides a system service object for the dynamic management of object instances. This is known as the ***ObjectManager*** object, and has operations defined on it for the run-time creation and deletion of object instances, the registration of wellknown capabilities, and the creation of replicated objects.

The replication-oriented operations are a temporary feature in Release 1 of Alpha, that were added in support of some initial application work. These operations do not constitute a proper set of mechanisms, but rather form a complete (if somewhat limited) replication facility. This replication facility includes specific object placement and replica management policies (i.e., a maximum of one replica per node, simple replica regeneration algorithm, and a primary plus backup copy scheme). A different set of operations will be provided in subsequent releases of Alpha; the object placement function belongs above the kernel level, and an additional, higher-level facility is required for the complete management replicated objects.

The interface to the ***ObjectManager*** object is shown in Figure 2, and its operations are described in the following text.

```
SYSTEM OBJECT    ObjectManager {
        OPERATION    Create(IN CAPA objectType,
                            OUT CAPA instance)
        OPERATION    Delete(IN CAPA instance)
        OPERATION    RegisterWellknown( IN CAPA regularCapa,
                                        IN CAPA wellknownCapa)
        OPERATION    ReplicatedCreate(IN CAPA objectType,
                                      IN u_long desiredReplicationFactor,
                                      OUT CAPA instance,
                                      OUT u_long actualReplicationFactor)
        OPERATION    Anchor()
}
```

Figure 2: Object Manager Object

Operation

Create(**IN CAPA** objectType,
OUT CAPA instance)

This operation is used to create a new instance of an object type from a given object type specification. The 'objectType' parameter is a capability for the object type specification for the type of object that is to be created by this operation, and 'instance' is a capability for the newly created object instance, and is returned when the operation has completed.

The object instance created by this operation is placed local to the node where the operation is performed. The full complement of rights are associated with the capability returned for the new object. This operation will fail if the specified object type is unknown, or if the node currently lacks the resources necessary to support another object.

Operation

Delete(**IN CAPA** instance)

This operation is used to delete an existing object instance. The 'instance' parameter is a capability for the object to be deleted.

If the passed capability is valid, this operation deletes the specified object and deallocates the resources associated with it. The indicated object is deleted regardless of its physical location in the system.

Operation

RegisterWellknown(**IN CAPA** regularCapa,
IN CAPA wellknownCapa)

This operation creates an alias for a given capability. The 'regularCapa' parameter is a capability that exists in the local dictionary, while the 'wellknownCapa' parameter is the wellknown capability that is to be bound to the entity indicated by the previous parameter.

Operation

ReplicatedCreate(**IN CAPA** objectType,

IN u_long desiredReplicationFactor,
OUT CAPA instance,
OUT u_long actualReplicationFactor)

This operation performs a function similar to that of the CREATE operation defined above, however it creates multiple instances of the new object. The 'objectType' parameter is a capability for the object type specification, and the 'instance' parameter is the returned capability for the newly instantiated (replicated) object. Furthermore, the 'desiredReplicationFactor' parameter is an integer that defines the (maximum) number of replicas that should be created, and the 'actualReplicationFactor' parameter is returned to indicated the actual number of replicas that were created.

This operation creates a replicated object instance, i.e., an object that consists of a number of replicas and a primary copy. The returned capability refers to the primary copy and does not change across the life of the replicated object (even when another replica takes over as the primary copy).

There are several reasons that the number of replicas created may differ from the number requested—e.g., there may not be sufficient resources for the requested number of replicas, and the replication policy dictates that there can be at most one replica per node.

Operation

Anchor()

This operation causes a thread to block on a semaphore (and otherwise do nothing). This operation is used in extending "lifeline" threads between objects to detect node/communications failures. When an invocation of this operation returns, the node on which this object exists can be considered to have failed.

3.1.2 Thread Manager

In support of threads, the kernel provides an object for the dynamic management of threads. This object is known as the ***ThreadManager*** object, and it provides functionality for threads that is analogous to what the previously defined ***ObjectManager*** object provides for objects.

The interface to the ***ThreadManager*** object is shown in Figure 3, and its operations are described in the following text.

```
SYSTEM OBJECT    ThreadManager {
        OPERATION    Create( IN CAPA instance,
                             IN u_long initialOperation,
                             IN parmblk initialParameters,
                             OUT CAPA thread)
        OPERATION    Delete(IN CAPA thread)
}
```

Figure 3: Thread Manager Object

Operation

Create(**IN CAPA** instance,
IN u_long initialOperation,
IN parmblk initialParameters,
OUT CAPA thread)

This operation is used to create threads. The parameters passed to this operation are a capability for the object (i.e., 'instance'), the name of the operation (i.e., 'initialOperation') that the newly created thread should begin execution within, and the initial parameters to be passed into the initial object. This operation returns a capability for the newly created thread (i.e., 'thread'), that contains all of the rights associated with threads. If the specified object does not currently exist on the node at which this operation is invoked, the operation fails.

Operation

Delete(**IN CAPA** thread)

This operation is used to delete a specified thread. A capability for a thread is given as the 'thread' parameter. Provided that the given capability is valid, this operation deletes the specified thread and deallocates the resources that are currently associated with it. Furthermore, the thread specified by the given capability is deleted regardless of its current location in the system. In addition, all of the thread's outstanding exception blocks are executed when a thread is deleted.

3.1.3 Semaphore Manager

The kernel provides the ***SemaphoreManager*** object in order to permit the dynamic creation and deletion of individual instances of ***Semaphore*** objects. In Release 1 of Alpha, ***Semaphore*** objects are bound to their creating object—i.e., each newly created ***Semaphore*** object can only be used by the object that created it, and likewise a ***Semaphore*** object may only be deleted by its creator.

The interface to the ***SemaphoreManager*** object is shown in Figure 4, and its operations are described in the following text.

```
SYSTEM OBJECT    SemaphoreManager {
        OPERATION    AllocSem(IN u_long count,
                              IN boolean pvUndo,
                              IN boolean allocUndo,
                              OUT CAPA Semaphore semaphore)
        OPERATION    DeallocSem(IN CAPA Semaphore semaphore)
}
```

Figure 4: Semaphore Manager Object

Operation

AllocSem(**IN** u_long count,
IN boolean pvUndo,
IN boolean allocUndo,
OUT CAPA Semaphore semaphore)

This operation instantiates an object of the kernel-defined type ***Semaphore***. This operation takes in a parameter to indicate the initial value that the semaphore's counter should take on (i.e., the 'count' parameter). It also takes a pair of boolean parameters that indicate whether the outstanding P and V operations should be undone on exceptions, and whether the allocation should be undone on exceptions (i.e., 'pvUndo' and 'allocUndo', respectively). These parameters are needed in order to permit the system's default exception handling code to restore an object's semaphores to their correct state should an exception occur while a thread is executing within it.

Furthermore, this operation returns a capability for the newly created ***Semaphore*** object to the invoking object in the 'semaphore' parameter. This capability has the NO_TRANSFER and NO_COPY restrictions applied to it (this is to ensure that semaphores and the objects that use them remain physically co-located with respect to each other).

Operation

DeallocSem(**IN CAPA** Semaphore semaphore)

This operation deletes the ***Semaphore*** object whose capability is passed as the 'semaphore' parameter to this operation. Because the NO_TRANSFER and NO_COPY restrictions are associated with the capabilities given to the ***Semaphore*** object's creator, only the object that created a semaphore has the ability to perform this operation on it.

3.1.4 Lock Manager

The ***LockManager*** object is provided by the kernel as a means of creating and deleting individual instances of ***Lock*** objects at run-time. In Release 1 of Alpha, locks are advisory—i.e., the underlying hardware does not enforce the desired synchronization regime, rather programmers must adhere to the proper conventions in order for locks to be effective. Because locks in Release 1 of Alpha are associated with contiguous regions of memory within an object, ***Lock*** objects, like ***Semaphore*** objects, remain local to the object that created them. Furthermore, the data regions associated with allocated locks are not permitted to overlap, therefore the ***LockManager*** returns a failure indication when an attempt is made to allocate a lock that intersects with an existing lock's data region.

The interface to the ***LockManager*** object is shown in Figure 5, and its operations are described in the following text.

```
SYSTEM OBJECT    LockManager {
        OPERATION    AllocLock(IN vaddr start,
                               IN u_long size,
                               OUT CAPA Lock lock)
        OPERATION    DeallocLock(IN CAPA Lock lock)
}
```

Figure 5: Lock Manager Object

Operation

AllocLock(**IN** vaddr start,
IN u_long size,
OUT CAPA Lock lock)

This operation is used to instantiate and initialize a ***Lock*** object that is associated with a data region within the object from which the operation is invoked. The data region that the lock is to cover is defined by the starting virtual address given by the 'start' parameter, and a byte count given by the 'size' parameter. A capability for the newly allocated ***Lock*** object is returned in the 'lock' parameter. As with ***Semaphore*** objects, the returned capability to the ***Lock*** object has the NO_TRANSFER and NO_COPY restrictions applied to it (ensuring that a lock and the object it is bound to remain physically co-located with respect to each other).

<u>Operation</u>

DeallocLock(**IN CAPA** Lock lock)

This operation is used to delete a lock and return to the system any resources associated with the ***Lock*** object indicated by the 'lock' parameter. Any threads that are blocked waiting to gain access to the deleted lock are unblocked.

3.2 Support Objects

The kernel provides other system service objects to support the Alpha kernel's abstractions. Included among these are objects of the system-defined object types for thread synchronization—i.e., ***Semaphore*** and ***Lock*** objects. Instances of these types of objects are generated by their respective manager objects (described in the preceding section). In addition, the kernel supplies a system service object that provides real-time clock services to the client. All of these objects are described in the following subsections.

3.2.1 Semaphore

Semaphore objects are instances of a kernel-defined object type, that are created and deleted by the ***SemaphoreManager*** object. Instances of ***Semaphore*** objects are used to control the concurrent execution of threads within objects. A ***Semaphore*** object provides functionality equivalent to the counting semaphore constructs found in many other systems.

Concurrency control is achieved by blocking and unblocking threads with these semaphores. In addition to providing the basic P and V operations, the ***Semaphore*** object has an operation defined on it that permits all of the threads that are blocked on the semaphore to be released. The latter operation is useful in a number of cases, including exception handling. The interface to the ***Semaphore*** object is shown in Figure 6, and its operations are described in the following text.

```
SYSTEM OBJECT     Semaphore {
        OPERATION    PSem()
        OPERATION    VSem()
        OPERATION    VAllSem()
}
```

Figure 6: Semaphore Object

<u>Operation</u>

PSem()

This operation performs a standard 'P' operation on a counting semaphore. Logically, this operation can be considered to represent an attempt at obtaining a token that corresponds to a resource that the ***Semaphore*** object manages. If a token is not available, the thread that invokes this operation is blocked until one becomes available. Once a token has been granted and the resource used, the thread then invokes a 'V' operation to, logically, return the synchronization token.

<u>Operation</u>

VSem()

This operation performs the standard 'V' operation on a counting semaphore. Logically, this operation represents the return of a previously granted token to the semaphore's resource pool. If there are any threads blocked waiting on a token, one of the threads that are blocked on the semaphore is granted the token and unblocked. The policy for selecting which of a set of blocked threads to release is definable at system build time. The policy currently in use is compatible with the best-effort scheduling policy used in Alpha.

Operation

VAllSem()

This operation unblocks all threads which may be waiting on the ***Semaphore*** object, and sets the semaphore's count to zero. If there are no threads queued on the semaphore when this operation is invoked, this operation has no effect.

3.2.2 Lock

Lock objects are instances of a kernel-defined object type that are created and deleted by the ***LockManager*** object. Instances of ***Lock*** objects are used to control the concurrent execution of threads within objects. Whereas ***Semaphore*** objects control the concurrent execution of threads within regions of code, the ***Lock*** objects control the concurrent access of threads to regions of memory within an object.

Lock objects are associated with specific data regions in an object, and threads executing within an object indicate their intentions to access the data covered by locks through the invocation of LOCK operations on the them. When a thread's manipulation of a locked data item is complete, the UNLOCK operation is invoked to indicate this fact to the kernel.

On LOCK operations, the permission to proceed with the desired operation on the region of data associated with the lock is determined by the system's locking policy (as manifest in the kernel's *lock compatibility table*). Threads are blocked until the lock can be granted in the requested mode. The locking policy currently in use by Release 1 of Alpha has the following modes: *exclusive read, concurrent read, exclusive write, concurrent write, and exclusive read/write*. The lock compatibility table for this policy is shown in Table 1. Furthermore, the unblocking of multiple queued, compatible, requests is performed in accordance with the system's time-driven resource management policy.

Requested \ Granted	Concurrent Read	Concurrent Write	Exclusive Read	Exclusive Write	Exclusive Read/Write
Concurrent Read	Compatible	Compatible	Not Compatible	Compatible	Not Compatible
Concurrent Write	Compatible	Compatible	Compatible	Not Compatible	Not Compatible
Exclusive Read	Not Compatible	Compatible	Not Compatible	Compatible	Not Compatible
Exclusive Write	Compatible	Not Compatible	Compatible	Not Compatible	Not Compatible
Exclusive Read/Write	Not Compatible	Not Compatible	Not Compatible	Not Compatible	Not Compatible

— Compatible — Not Compatible

Table 1: Lock Compatibility Table

When a thread completes execution within an object, the system automatically releases all of the locks that it holds (just as is done with semaphore's). In addition, the default exception handler restores the state of all locks held in *write* modes by a thread that encounters an exception, it also releases all locks acquired by the thread within the exception block.

The interface to the ***Lock*** object is shown in Figure 7, and its operations are described in the following text.

```
SYSTEM OBJECT    Lock {
        OPERATION    Lock(IN u_long mode)
        OPERATION    ConditionalLock(IN u_long mode,
                                     OUT boolean result)
        OPERATION    UnLock()
        OPERATION    Convert(IN u_long newMode)
        OPERATION    Query(OUT u_long currentMode)
}
```

Figure 7: Lock Object

<u>Operation</u>

Lock(**IN** u_long mode)

This operation is used to acquire a lock—i.e., to announce to the kernel an intention on the part of the invoking thread to access the region of data associated with the lock, in a manner indicated by the 'mode' parameter. If the locking policy that is in place in the system indicates that the requested mode is compatible with the mode that the lock is currently in, then the operation returns and the thread can proceed to manipulate the locked data in the indicated manner. However, if the operation is determined to be incompatible with the current state of the lock, the invoking thread is blocked until a time when the request becomes compatible.

Operation

ConditionalLock(**IN** u_long mode,
OUT boolean result)

This operation is identical in function to the previously described operation, with the exception that the thread that invokes this operation is never blocked. Rather, a boolean flag is returned in the 'result' parameter, a logical TRUE indicates that the lock has been successfully acquired, and FALSE indicates that the lock could not be acquired in the requested mode.

Operation

UnLock()

This operation is used to release a previously acquired lock. If the invoking thread does not hold the lock (in any mode) when this operation is invoked, the operation has no effect.

Operation

Convert(**IN** u_long newMode)

This operation allows a thread to change the mode in which a lock is held to the mode indicated by the 'newMode' parameter. This operation behaves much the same as the LOCK operation. If the lock is already being held by the thread, this operation does not require that the lock be released first and then reacquired in the new mode, instead it makes these two operations appear as though they were performed atomically. If the lock is not being held at the time this operation is invoked, this acts exactly the same as a LOCK operation.

Operation

Query(**OUT** u_long currentMode)

This operation is used by a thread to obtain the state of a lock at the time that the operation is executed. The lock state is returned in the 'currentMode' parameter.

3.2.3 Time Services

This object offers time-related services to clients. Included in these services is a reading of the value of the local node's real-time clock, and an operation that can be used to suspend the execution of a thread until (at least) a specified amount of (real) time has passed. The interface to the ***TimeServices*** object is shown in Figure 8, and its operations are described in the following text.

```
SYSTEM OBJECT   TimeServices {
        OPERATION   GetTime(OUT u_long timeval)
        OPERATION   Sleep(IN u_long delay)
}
```

Figure 8: Time Services Object

Operation

GetTime(**OUT** u_long timeval)

This operation reads the system clock, and returns an unsigned integer value which represents the current time (in units of 200 nanoseconds). The time value is returned in the 'timeval' parameter.

Operation

Sleep(**IN** u_long delay)

This operation suspends the execution of the invoking thread for (at least) a period of time indicated by the 'delay' parameter. In Release 1 of Alpha, the 'delay' parameter is an unsigned integer that represents the amount of time the thread has to be delayed in terms of milliseconds.

3.3 I/O Objects

Release 1 of the Alpha kernel also provides a pair of system service objects that provide the client with the ability to perform I/O with machines external to the testbed's execution environment. See [Northcutt 88c] for a description of the overall testbed environment.

These I/O system service objects provide the basic interfaces necessary for a node to communicate over the application processor's serial interface line, and the node's network connection. The following subsections provide a description of each of these objects and their interfaces.

3.3.1 TTY

The ***TTY*** system service object provides a simple send/receive interface to a serial line unit on a testbed node's application processor. The object is capable of performing functions equivalent to the "raw" and "cooked" modes of serial I/O found in UNIX. This is to say that the ***TTY*** object can be used to perform either direct (i.e., unbuffered, single-character) I/O, or buffered I/O with line editing (i.e., complete, carriage-return-terminated strings are returned).

The mode that the ***TTY*** object operates in is defined by invoking an operation on the object, and it can be changed at run-time. This object only provides the basic I/O functions—functions such as reliable block transfers are intended to be implemented by higher-level objects that make use of this one.

The interface to the ***TTY*** object is shown in Figure 9, and its operations are described in the following text.

```
SYSTEM OBJECT    TTY {
        OPERATION    Init(IN u_long mode)
        OPERATION    Send(IN char msg)
        OPERATION    Receive(OUT char msg)
}
```

Figure 9: TTY Object

Operation

Init(**IN** u_long mode)

This operation initializes the object and defines the mode of operation based on the value of the 'mode' parameter. Encoded in this parameter is the desired baud rate, and the mode (i.e., either *raw* or *cooked*).

Operation

Send(**IN** char msg[])

This operation takes a character string (i.e., the 'msg' parameter), and sends it to the serial port encapsulated by the ***TTY*** object. In raw mode this operation returns when the characters have all been transmitted, while in cooked mode the operation returns once the characters have been buffered for output.

Operation

Receive(**OUT** char msg[])

This operation waits for characters to arrive on the object's encapsulated serial port, and then returns them in the 'msg' parameter. In raw mode, a single character is returned, while in cooked mode, the operation does not return until it has received a full input line (i.e., a carriage-return-terminated string). In either case, the invoking thread blocks until characters are available.

3.3.2 Network I/O

This system service object provides an extremely simple message transport interface to the testbed's communications interconnect (i.e., Ethernet). A simple (IP/UDP-based) communications protocol is used to permit short (i.e., less than 256 byte) messages to be exchanged, across the testbed network, with a UNIX process on a gateway machine. This object provides basic, reliable block I/O functions, and higher-level functions can be built on top of it.

The interface to the ***NetIO*** object is shown in Figure 10, and its operations are described in the following text.

```
SYSTEM OBJECT    NetIO {
        OPERATION    Init()
        OPERATION    Send(IN extmsg msg)
        OPERATION    Receive(OUT extmsg msg)
}
```

Figure 10: Network I/O Object

Operation

Init()

This operation initializes the network I/O object. Following the invocation of this operation a node is capable of sending and receiving messages from the gateway machine. This operation need only be executed once (typically at node start-up).

Operation

Send(**IN** extmsg msg)

This operation takes a message (in the 'msg' parameter), queues it for transmission, and then returns to the invoker.

Operation

Receive(**OUT** extmsg msg)

This operation attempts to dequeue a message and place it into the 'msg' parameter. The invoking thread is blocked until a message arrives if there is not one immediately available.

3.4 Miscellaneous Object

As a temporary measure in Release 1, a number of necessary kernel functions have been grouped together into a "catch-all", system service object. This is known as the ***TempServices*** object, and provides clients with the ability to manage wellknown capabilities, make a thread ready for execution, modify the importance of a thread, and enable the scheduler's run-time statistics gathering facility.

Many of these operations will be subsumed into the thread and object standard operations set in Release 2.

The interface to the ***TempServices*** object is shown in Figure 11, and its operations are described in the following text.

```
SYSTEM OBJECT    TempServices {
        OPERATION    AddWellknown(IN CAPA instance,
                                  IN CAPA wellknown)
        OPERATION    MakeReady(IN CAPA thread)
        OPERATION    SetThreadImportance(IN CAPA thread,
                                         IN u_long new_importance,
                                         OUT u_long old_importance)
        OPERATION    SchedStats(IN boolean switch)
}
```

Figure 11: Temporary Services Object

Operation

AddWellknown(**IN CAPA** instance,
IN CAPA wellknown)

This operation is used to add the wellknown capability given in the 'wellknown' parameter to the c-list of the object indicated by the 'instance' parameter.

Operation

MakeReady(**IN CAPA** thread)

This operation unblocks a newly created thread (indicated by the capability given by the 'thread' parameter) and makes it ready to run. This operation must be invoked on a newly created thread to start it executing. When a new thread is created it is blocked and cannot begin execution until this operation is invoked.

Operation

SetThreadImportance(**IN CAPA** thread,
IN u_long newImportance,
OUT u_long oldImportance)

This operation is used to read and modify the importance value of an existing thread. The thread to be modified is given by the capability in the 'thread' parameter, the new importance is an integer value given in the 'newImportance' parameter, and the value of the thread's importance prior to this change is returned in the 'oldImportance' parameter.

Operation

SchedStats(**IN** boolean switch)

This operation enables and disables the scheduling subsystem's statistics gathering facility. The facility is enabled when the 'switch' parameter is TRUE, and is disabled when it is FALSE.

References

[Clark 88] Clark, R. K., Kegley, R. B., Keleher, P. J., Maynard, D. P., Northcutt, J. D., Shipman, S. E. and Zimmerman, B. A.
An Example Real-Time Command and Control Application.
Archons Project Technical Report #88032, Department of Computer Science, Carnegie-Mellon University, March, 1988.

[Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer Science, Carnegie-Mellon University, April, 1988.

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January, 1988.

[Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model.
Archons Project Technical Report #88021, Department of Computer Science, Carnegie-Mellon University, February, 1988.

[Northcutt 88c] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie-Mellon University, March, 1988.

[Northcutt 88d] Northcutt, J. D. and Shipman, S. E.
The Alpha Operating System: Programming Utilities.
Archons Project Technical Report #88061, Department of Computer Science, Carnegie-Mellon University, June, 1988.

Appendix I: Kernel Interface Usage Examples

The follow is a source code listing of the output generated by the Release 1 object-language preprocessor. The intent of this example is to illustrate the means by which operations can be invoked from client objects.

This example was drawn from the joint CMU/GD air-defense application [Clark 88], and shows the object language source code (as defined in [Shipman 88]), as well as the output of the object language preprocessor. This example illustrates the details of how the kernel mechanisms are used by showing how the preprocessor makes use of them.

Object Source Code Example

```
/*******************************************************************
 *                                                                 *
 *                   Distributed C2 Demonstrator                   *
 *                                                                 *
 *               General Dynamics Fort Worth Division              *
 *              Carnegie-Mellon University Archons Project         *
 *                                                                 *
 *******************************************************************/



OBJECT InterfaceInit {
/*************************************************************************\
*
* System Interface Node Initialization Object
* ---------------------------------------------
* Creates and intializes the objects which run on the system interface node.
*
\*************************************************************************/


/*
* Include Files
*/

#include    "../include/defs.h"
#include    "../include/c2demo.h"
#include    "../include/serialio.h"


/*
* Data Declarations
*/

WELLKNOWN CAPA CommIntfTypeComm;
WELLKNOWN CAPA DistrSvcMgrTypeDSM;

OPERATION Init()
{
    u_int       size;
    status_msg  stat;           /* Status message buffer */
    time_type   StartTime = 0;
    short       num_appl_nodes; /* count passed in by EXPCONTROL */
    dsm_mask_t  dsm_masks;      /* DSM capability masks */

    /* Initialize SerialIO */
    INVOKE Comm.Init(OUT (short) num_appl_nodes, OUT (dsm_mask_t) dsm_masks)
    ON FAILURE {
        ERROR_HALT();
    };

    sprintf(stat.buffer,"Comm inited");
    size = (STAT_MSG_HDR + strlen(stat.buffer));
```

```
    UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                  IN (u_int)size,
                                  IN (status_msg)stat);

    /* initialize 'DistrSvcMgr' -- also sends Alpha Ready */
    INVOKE DSM.Init(IN (short) num_appl_nodes, IN (dsm_mask_t) dsm_masks)
    ON FAILURE {
        sprintf(stat.buffer,"DSM init failed");
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg)stat);
        ERROR_HALT();
    }

    sprintf(stat.buffer,"DSM inited");
    size = (STAT_MSG_HDR + strlen(stat.buffer));
    UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                  IN (u_int)size,
                                  IN (status_msg)stat);
}
}
```

Preprocessor Output Example

```
asm("   .data");
```

```
asm("   .even");
asm("   .globl  _OCC_OBJ_InterfaceInit");
asm("_OCC_OBJ_InterfaceInit:  .long     1");
asm("   .long   _OCC_OP_InterfaceInit_Init");
#define SUCCESS         0
#define FAILURE         1
#define SYSTEM_FAILURE  2
#define RESULT          (OCC_orply->hdr.opresult)
#define STATUS          (OCC_orply->hdr.opstatus)
#define OCC_PARM_MAX    (2048-sizeof(OCC_rqsthdr)-sizeof(OCC_rplyhdr))
typedef struct {
    unsigned short   objindex;
    unsigned short   oprindex;
    unsigned short   numparms;
    unsigned short   numcapas;
    unsigned long    senderid;
    unsigned long    recipientid;
    unsigned long    threadid;
} OCC_rqsthdr;
typedef struct {
    unsigned short   opresult;
    unsigned short   opstatus;
    unsigned short   numparms;
    unsigned short   numcapas;
    unsigned long    senderid;
    unsigned long    recipientid;
    unsigned long    threadid;
    unsigned long    importance;
    unsigned short   tcflag;
    unsigned long    tolct;
    unsigned long    rdeadline;
    unsigned long    rreqtime;
    unsigned long    meanct;
    unsigned long    stddevct;
    unsigned short   iVearly;
    unsigned short   iVlate;
    unsigned short   ablklevel;
    float            xasf;
    float            yasfE;
    float            yasfL;
} OCC_rplyhdr;
typedef struct {
    OCC_rqsthdr      hdr;
    unsigned long    parms[OCC_PARM_MAX];
} *OCC_parmblkptr;
typedef struct {
    OCC_rplyhdr      hdr;
    unsigned long    parms[OCC_PARM_MAX];
} *OCC_rplypblkptr;


#undef OCC_PARM_MAX
typedef struct {
    float                  deadline;    /* in milliseconds */
    float                  meanct;      /* mean computation time */
    float                  stddevct;    /* standard deviation for meanct */
    unsigned short         iVearly;     /* index for early value func */
```

```
    unsigned short         iVlate;       /* index for late value function */
    short                  imptmod;      /* importance modifier in % */
    short                  imptdrop;     /* % impt drop after crit time */
    unsigned short         TCFrameSeq;   /* TC frame sequence */
} OCC_DL_tcspb;
typedef struct {
    unsigned long    apthreadId;
    unsigned long    spthreadId;
    unsigned long    threadabid;
    unsigned short   ablklevel;
    union {
         unsigned long     cspb;
         OCC_DL_tcspbt     cspb;
    } spb;
} OCC_DL_sib;
static unsigned longComm = 0;
static unsigned longDSM = 1;

/***************************************************************************\
*
* System Interface Node Initialization Object
* -------------------------------------------
* Creates and intializes the objects which run on the system interface node.
*
\***************************************************************************/

#include    "../h/c3demo.h"
#include    "../h/defs.h"
#include    "../h/serialio.h"

void OCC_OP_InterfaceInit_Init(OCC_ipbp, OCC_opbp)
    register OCC_parmblkptr OCC_ipbp, OCC_opbp;
{
    OCC_rplypblkptrOCC_irply, OCC_orply;
    OCC_DL_sib  OCC_sib;
    OCC_irply = ((OCC_ipbp == 0) ?
        0 :
        (OCC_rplypblkptr)((char *)OCC_ipbp + sizeof(OCC_rqsthdr) +
        ((OCC_ipbp->hdr.numparms+OCC_ipbp->hdr.numcapas) * sizeof(long))));
    {                                   /* Get Incoming Parameters*/
        if (OCC_irply != 0) {
            OCC_irply->hdr.opresult = 1;   /* no RETURN is FAILURE */
            OCC_irply->hdr.numparms = 0;
            OCC_irply->hdr.numcapas = 0;
            OCC_irply->hdr.opstatus = 0;
        }
    }
   { int i=0; if (i) goto OCC_label; } /* prevent 'stmt not reached' */
{
    u_int        size;
    status_msg   stat;            /* Status message buffer */
    time_type    StartTime = 0;
    short        num_appl_nodes; /* count passed in by EXPCONTROL */
    dsm_mask_t   dsm_masks;      /* DSM capability masks */

    /* Initialize SerialIO */
```

```
{
    char *OCC_parmptr; unsigned long OCC_capaindex;
    OCC_parmptr = (char *)&(OCC_opbp->parms[0]);
    OCC_opbp->hdr.objindex = 0;/* Comm */
    {    extern unsigned long OCC_OO_Comm_WKC_Init;
         OCC_opbp->hdr.oprindex = OCC_OO_Comm_WKC_Init;    }
    OCC_capaindex=(unsigned long)(OCC_parmptr-(char *)&(OCC_opbp->parms[0]));
    OCC_opbp->hdr.numparms = (OCC_capaindex % sizeof(long)) ?
        OCC_capaindex/sizeof(long) + 1 : OCC_capaindex/sizeof(long);
    OCC_capaindex = 0;
    OCC_opbp->hdr.numcapas = OCC_capaindex;
    asm("   trap #10");     /* PERFORM INVOKE */
    OCC_orply =
      (OCC_rplypblkptr)((char *)OCC_opbp + sizeof(OCC_rqsthdr) +
      ((OCC_opbp->hdr.numparms+OCC_opbp->hdr.numcapas) * sizeof(long)));
    if (OCC_orply->hdr.opresult == 0) {
        OCC_parmptr = (char *)&(OCC_orply->parms[0]);
        OCC_capaindex = 0;
        num_appl_nodes = *((short *)OCC_parmptr);
        OCC_parmptr += (sizeof(short) & 0x1) ?
                            sizeof(short)+1 : sizeof(short);
        dsm_masks = *((dsm_mask_t *)OCC_parmptr);
        OCC_parmptr += (sizeof(dsm_mask_t) & 0x1) ?
                            sizeof(dsm_mask_t)+1 : sizeof(dsm_mask_t);
    }
}
if (OCC_orply->hdr.opresult != 0) {
    ERROR_HALT();
}
sprintf(stat.buffer,"Comm inited");
size = (STAT_MSG_HDR + strlen(stat.buffer));
{
    char *OCC_parmptr; unsigned long OCC_capaindex;
    OCC_parmptr = (char *)&(OCC_opbp->parms[0]);
    OCC_opbp->hdr.objindex = 0;  /* Comm */
    {    extern unsigned long OCC_OO_Comm_WKC_SendMsg;
         OCC_opbp->hdr.oprindex = OCC_OO_Comm_WKC_SendMsg;    }
    *((msg_t *)OCC_parmptr) = MSG_ALPHA_STATUS;
    OCC_parmptr += (sizeof(msg_t) & 0x1) ? sizeof(msg_t)+1 : sizeof(msg_t);
    *((u_int *)OCC_parmptr) = size;
    OCC_parmptr += (sizeof(u_int) & 0x1) ? sizeof(u_int)+1 : sizeof(u_int);
    *((status_msg_t *)OCC_parmptr) = stat;

    OCC_parmptr += (sizeof(status_msg) & 0x1) ?
                    sizeof(status_msg)+1 : sizeof(status_msg);
    OCC_capaindex=(unsigned long)(OCC_parmptr-(char *)&
                                  (OCC_opbp->parms[0]));
    OCC_opbp->hdr.numparms =  (OCC_capaindex % sizeof(long)) ?
        OCC_capaindex/sizeof(long) + 1 : OCC_capaindex/sizeof(long);
    OCC_capaindex = 0;
    OCC_opbp->hdr.numcapas = OCC_capaindex;
    asm("   trap  #10");    /* PERFORM INVOKE */
    OCC_orply =
            (OCC_rplypblkptr)((char *)OCC_opbp + sizeof(OCC_rqsthdr) +
            ((OCC_opbp->hdr.numparms+OCC_opbp->hdr.numcapas)    *    sizeof(-
```

```
long)));
        if (OCC_orply->hdr.opresult == 0) {
            OCC_parmptr = (char *)&(OCC_orply->parms[0]);
            OCC_capaindex = 0;
        }
    }

    /* initialize 'DistrSvcMgr' -- also sends Alpha Ready */
    {
        char *OCC_parmptr; unsigned long OCC_capaindex;
        OCC_parmptr = (char *)&(OCC_opbp->parms[0]);
        OCC_opbp->hdr.objindex = 1;  /* DSM */
        {    extern unsigned long OCC_OO_DSM_WKC_Init;
             OCC_opbp->hdr.oprindex = OCC_OO_DSM_WKC_Init;    }

        *((short *)OCC_parmptr) = num_appl_nodes;
        OCC_parmptr += (sizeof(short) & 0x1) ? sizeof(short)+1 : sizeof(short);

        *((dsm_mask_t *)OCC_parmptr) = dsm_masks;
        OCC_parmptr += (sizeof(dsm_mask_t) & 0x1) ?
                        sizeof(dsm_mask_t)+1 : sizeof(dsm_mask_t);
        OCC_capaindex=(unsigned long)(OCC_parmptr-(char *)&(OCC_opbp->parms[0]));
        OCC_opbp->hdr.numparms =  (OCC_capaindex % sizeof(long)) ?
                    OCC_capaindex/sizeof(long) + 1 : OCC_capaindex/sizeof(long);
        OCC_capaindex = 0;
        OCC_opbp->hdr.numcapas = OCC_capaindex;
        asm("   trap  #10");    /* PERFORM INVOKE */
        OCC_orply =
                (OCC_rplypblkptr)((char *)OCC_opbp + sizeof(OCC_rqsthdr) +
                ((OCC_opbp->hdr.numparms+OCC_opbp->hdr.numcapas) * sizeof(long)));
        if (OCC_orply->hdr.opresult == 0) {
            OCC_parmptr = (char *)&(OCC_orply->parms[0]);
            OCC_capaindex = 0;
        }
    }
    if (OCC_orply->hdr.opresult != 0) {
        sprintf(stat.buffer,"DSM init failed");
        size = (STAT_MSG_HDR + strlen(stat.buffer));

    {
        char *OCC_parmptr; unsigned long OCC_capaindex;
        OCC_parmptr = (char *)&(OCC_opbp->parms[0]);
        OCC_opbp->hdr.objindex = 0;    /* Comm */
        {    extern unsigned long OCC_OO_Comm_WKC_SendMsg;
             OCC_opbp->hdr.oprindex = OCC_OO_Comm_WKC_SendMsg;    }
        *((msg_t *)OCC_parmptr) = MSG_ALPHA_STATUS;
        OCC_parmptr += (sizeof(msg_t) & 0x1) ? sizeof(msg_t)+1 : sizeof(msg_t);
        *((u_int *)OCC_parmptr) = size;
        OCC_parmptr += (sizeof(u_int) & 0x1) ? sizeof(u_int)+1 : sizeof(u_int);
        *((status_msg_t *)OCC_parmptr) = stat;
        OCC_parmptr += (sizeof(status_msg_t) & 0x1) ?
                        sizeof(status_msg_t)+1 : sizeof(status_msg_t);
        OCC_capaindex=(unsigned long)(OCC_parmptr-(char *)&(OCC_opbp->parms[0]));
        OCC_opbp->hdr.numparms = (OCC_capaindex % sizeof(long)) ?
```

```
                    OCC_capaindex/sizeof(long) + 1 : OCC_capaindex/sizeof(long);
        OCC_capaindex = 0;
        OCC_opbp->hdr.numcapas = OCC_capaindex;
        asm("   trap  #10");    /* PERFORM INVOKE */
        OCC_orply =
                (OCC_rplypblkptr)((char *)OCC_opbp + sizeof(OCC_rqsthdr) +
                ((OCC_opbp->hdr.numparms+OCC_opbp->hdr.numcapas) *
                                                        sizeof(long)));
        if (OCC_orply->hdr.opresult == 0) {
            OCC_parmptr = (char *)&(OCC_orply->parms[0]);
            OCC_capaindex = 0;
        }
    }
        ERROR_HALT();
    }

    sprintf(stat.buffer,"DSM inited");
    size = (STAT_MSG_HDR + strlen(stat.buffer));
    {
        char *OCC_parmptr; unsigned long OCC_capaindex;
        OCC_parmptr = (char *)&(OCC_opbp->parms[0]);
        OCC_opbp->hdr.objindex = 0;    /* Comm */
        {    extern unsigned long OCC_OO_Comm_WKC_SendMsg;
             OCC_opbp->hdr.oprindex = OCC_OO_Comm_WKC_SendMsg;    }
        *((msg_t *)OCC_parmptr) = MSG_ALPHA_STATUS;
        OCC_parmptr += (sizeof(msg_t) & 0x1) ? sizeof(msg_t)+1 : sizeof(msg_t);
        *((u_int *)OCC_parmptr) = size;
        OCC_parmptr += (sizeof(u_int) & 0x1) ? sizeof(u_int)+1 : sizeof(u_int);
        *((status_msg_t *)OCC_parmptr) = stat;
        OCC_parmptr += (sizeof(status_msg_t) & 0x1) ?
                        sizeof(status_msg_t)+1 : sizeof(status_msg_t);
        OCC_capaindex=(unsigned long)(OCC_parmptr-(char *)&(OCC_opbp->parms[0]));
        OCC_opbp->hdr.numparms = (OCC_capaindex % sizeof(long)) ?
                    OCC_capaindex/sizeof(long) + 1 : OCC_capaindex/sizeof(long);
        OCC_capaindex = 0;
        OCC_opbp->hdr.numcapas = OCC_capaindex;
        asm("   trap  #10");    /* PERFORM INVOKE */
        OCC_orply =
                (OCC_rplypblkptr)((char *)OCC_opbp + sizeof(OCC_rqsthdr) +
                ((OCC_opbp->hdr.numparms+OCC_opbp->hdr.numcapas)    *    sizeof(-
long)));
        if (OCC_orply->hdr.opresult == 0) {
            OCC_parmptr = (char *)&(OCC_orply->parms[0]);
            OCC_capaindex = 0;
        }
    }
}       /* falling through succeeds automatically */
OCC_label:
    {                                           /* RETURN SUCCESS */
        char *OCC_parmptr; unsigned long OCC_capaindex;
        OCC_parmptr = (char *)&(OCC_irply->parms[0]);
        OCC_irply->hdr.opresult = 0;
        OCC_irply->hdr.opstatus = 0;
        OCC_capaindex=(unsigned      long)(OCC_parmptr-(char      *)&(OCC_irply-
```

```
>parms[0]));
        OCC_irply->hdr.numparms =  (OCC_capaindex % sizeof(long)) ?
            OCC_capaindex/sizeof(long) + 1 : OCC_capaindex/sizeof(long);
        OCC_capaindex = 0;
        OCC_irply->hdr.numcapas = OCC_capaindex;
        return;
    }
}       /* END OF OPERATION */
```