NCR CONTROL DATA

ADVANCED SYSTEMS LABORATORY

CHAPTER 06

PROGRAM MANAGEMENT

Doc. No. ASL00282

Rev. 04

Copy No. 87

TABLE OF CONTENTS

1.0 INTRODUCTION

IPLOS Program Management provides the mechanisms through which the user may organize and present his programs to the system. The three basic constructs of Program Management are:

STATIC CONSTRUCT	DYNAMIC CONSTRUCT	CHARACTERISTICS	ANALOGIES
Job	Job	.Single address space .Batch submission or single user terminal session	Job in most systems
Program	Task	.Separate naming context (entry pts - externals) .Separate common block allocations .Separate load	COBOL Run Unit CYBER Program CENTURY Program PL3S Task MASTER Task OS/VS Job Step
Procedure	Subtask	.Separate stack frame	PL/I Task CENTURY B2 Task BURROUGHS Async Procedure

TABLE 1.0-1

PROGRAM MANAGEMENT BASIC EXECUTION CONSTRUCTS

The progression from job to task to subtask is characterized by a.) decreasing amounts of static data, b.) decreasing overhead involved in initiation, and c.) increasing amounts of automatically shared data.

Each of these constructs is dealt with in greater detail in the ensuing parts of this section.

Program Management also provides the mechanisms for communications between jobs and between programs in execution.

For communication between nonsimultaneously active jobs, a mailbox file is provided. The mailbox provides a permanent repository (i.e., unrelated to the life of a particular job), for messages. This enables jobs to enter the system in arbitrary order, at arbitrary times, and to sequence and synchronize their subsequent activations.

For executing jobs, tasks, or subtasks, the following communication mechanisms are available:

LNS
Signals
Queues
Events
Semaphores
Signature locks
On conditions

These mechanisms allow jobs, tasks, and subtasks to synchronize and coordinate themselves with other asynchronous activities. These mechanisms and the requests which are used to manipulate them are treated in greater detail in ensuing parts of this section.

TYPE	SCOPE	DATA	LIFETIME	USAGE
Mailbox	Inter Job	Arbitrary	Immortal	Job sequencing, Communication between users
LNS local	Intra Job	Predefined by type	Job	Symbolic access from terminal, Passing parameters from user to the system
LNS global	Inter Job		System	
Event	Intra Job	Boolean	(Job) LNS, Stack, Static	Synchronization, Interrupt control
Signal	Inter Job	128 bytes	DEQUEUE or Overwrite	I/O requests, System Job Communications
Queue	Intra Job	Arbitrary	(Job) LNS, Stack, Static	Queuing signals, passing data
Sema-phore	Intra Job	Integer	(Job) LNS, Stack, Static	Synchronization, Locking (using shared resources)
Sign Locks	Inter Job	Compare Swap word	Segment	Synchronization (Compare Swap)
On Con-dition	Intra Program	Condition Register	Stack	Handling execution condition. See Doc ASL00211.

TABLE 1.0-2
PROGRAM MANAGEMENT BASIC COMMUNICATION CONSTRUCTS

1.1 REQUIREMENTS AND OBJECTIVES

The following is a summary of the major requirements and objectives that motivate the design of IPLOS Program Management:

- o ANSI standard COBOL (excluding PLC proposal ATG-71001.11: Asynchronous Processing Facility)
- o ANSI standard FORTRAN
- o Multiprocessing-multiple degrees of sharing and overhead to initiate
- o Protection - multiple subsystem services in the same address space
- o Sharing - effective use of a large virtual memory

1.2 DEFINITION OF TERMS

The following are definitions of terms relevant to Program Management.

Address Space — The set of segments addressable in a job. Each address is uniquely identified by a segment number and a byte number.

Binary Object File — A file containing one or more contiguous object modules. All object modules in the segment have the same segment attributes.

Binding Section — The object environment component used to control transfer between rings of protection. There is one binding section per loaded module.

Binding Segment — A segment containing the binding sections of one or more modules loaded into the address space of a job. There are several binding segments per job.

Condition — A synchronous occurrence of interest to the task or subtask in which it occurred. The arithmetic faults, such as overflow, are examples of conditions.

Control Point — The basic execution entity recognized and

1.0 INTRODUCTION
1.2 DEFINITION OF TERMS

	dispatched by the System Monitor. Among its contents is the hardware defined Exchange Package.
Control Point id	A system unique identification of a control point used as the destination address of signals.
Entry Point	A named externally accessible address in a module. The entry points may be in either the code section or the working storage section of the module.
Event	An asynchronous occurrence of significance to a task or subtask. Task completion, time, and I/O completion are typical examples of events.
Event Control Block	A data structure required to manipulate the flow of control via event requests. May be in LNS, internal static, or a stack.
External	A symbol referenced by a module that is defined as an entry point in another module.
Gate	A hardware protected entry point for crossing between programs. Protection changes can only occur at gates. Validation of the right to change can be done at the gate.
Gate Registration	The act of making a gate known within a job, such that subsequent loading will link to the protected entry point when referenced.
Global Key	One of the two keys associated with every known segment. Verified on every access and on call/return sequences. Intended as a mechanism for isolating programs executing in the same ring of protection. Not supported in v 1.0.
Job	Job is defined in Section 1.0 of Chapter 4 of the OSGDS.

1.0 INTRODUCTION
1.2 DEFINITION OF TERMS

Job Gate Table	A table used by Program Management to register gated entry points on a job basis.
Job Stack Table	A table used by Program Management for ring by ring allocation of stacks when a control point is created.
Library	A segment containing procedures and the dictionaries required to locate them. The procedure dictionary is organized by entry point name. All load modules in the library have the same segment attributes.
Load Module	An object module reformatted by OBLIGE for residency on a library. Can be a single procedure. Structured as directly referenceable storage; code section shareable among users.
Loader Map	The output of the Loader describing the allocations performed for all the sections of all the modules in the loaded program.
Loader Symbol Table	An internal table built and used by the Loader for matching externals and entry points. There is a separate Loader Symbol Table per loaded program.
Local Key	One of the two keys associated with every known segment. Verified on every access. Always associated with the segment and not verified or passed on by call/return sequences.
Mailbox	A file used for communication between two users, for example, for job sequencing. May contain messages.
Object Module	A single piece of machine executable code output from a compiler. Structured as a series of records on a file that are interpreted every time the object module is processed.
Procedure	Code that may be executed serially via

1.0 INTRODUCTION
1.2 DEFINITION OF TERMS

hardware call instruction or executed asynchronously via spawning a subtask.

Program — A set of object files, set of libraries, and an entry point name which specifies a static set of procedures organized to perform some specific function (e.g., compile COBOL statements). An activation of a program is a task.

Program Control Block — LNS structure required to construct a program by linking external references and entry points in a specified order. It can be in any LNS segment.

Queue — A collection of data items awaiting processing. Standard signals are queued.

Queue Control Block — A data structure required to manipulate a queue via queue requests. May be in LNS, internal static, or a stack.

Ring — The fifteen hierarchial levels of protection available within a single job. Used to protect local monitors and services from their users. Capability in ring n is always greater than or equal to capability in ring n+1.

Semaphore — A system supported facility to permit synchronization among asynchronous activities within a job. It is the most primitive such facility supported by the system.

Signal — A signal is a short message primarily used for inter-job communications in the form of requests and responses.

Signal Buffer — A system structure used to interface signal reception by a control point.

Signal Selection List — A system table used to register signal selections on a control point basis.

Signature Lock — The externalization of Compare-Swap for locking data in shared segments between

1.0 INTRODUCTION
1.2 DEFINITION OF TERMS

jobs.

Subsystem — A job which provides services to the user in the same way as those provided by the System Job. It is protected from the user, and the Operating System is protected from it.

Subsystem Services — A set of shared procedures (both code and internal static) which provide Subsystem services and are directly callable. They have the same clock accounting, scheduling and execution characteristics as the requestor. The only difference is their access rights to data and code. They are also protected from Task Services, that is, in a different ring.

Subtask — Asynchronous execution of a procedure within a single task. All static data associated with the task is associated with the subtasks. The subtask receives only a new stack segment as a repository for private data.

Task — Identifiable execution of a program.

Task Control Block — A system LNS data structure required to identify a task and pass it parameters.

Task Monitor — A collection of shared, nonresident, reentrant procedures which monitor and provide a formal interface between user and system monitor.

Task Services — A set of shared procedures (both code and internal static) which provide Operating System services and are directly callable. They have the same clock accounting, scheduling and execution characteristics as the requestor. The only difference is their access rights to data and code.

2.0 PROGRAM EXECUTION

2.1 EXECUTION CONSTRUCTS

IPLOS supports three major execution constructs:

- o JOB
- o TASK
- o SUBTASK

2.1.1 JOB

The job is the mechanism through which the batch or interactive user interfaces to the IPL system. A job consists of a single segmented address space and all the work performed by the job takes place within that address space.

The convention of associating a single address space with a job is not mandatory, however, the OS project feels that there are several factors which make it desirable:

- o It allows natural sharing of information between components of the job - all information is addressed through the same mechanism (i.e., the same segment descriptor table)

- o It allows the code which manages the components of a job (i.e., program establisher, task establisher, loader) to be a part of the same job thereby a.) facilitating the component management and b.) isolating it from other jobs and the system code responsible for job management.

- o It allows large amounts of the system and user provided environment that all components of the job depend upon to only be established once for all the components in the job (e.g., task monitor, subsystem services, etc.)

- o It allows straightforward invocation and parameter passing between the aforementioned shared environment and a user task.

There are also several disadvantages to the single job per address space relationship:

- o The volatility of comings and goings of programs and data within the address space forces the loading of absolutized components to be preplanned (i.e., the permanent reservation of a segment in every address space).

- o Components that are independent of each other and have therefore no need to share or communicate are unprotected from each other and therefore subject to time dependent errors. This may be improved somewhat by utilizing the various protection mechanisms available within the address space (.e.g., rings, global or local keys).

In spite of these disadvantages, we feel that the single address space per job is the best way to proceed.

2.1.2 TASK

A program is the principal way work is organized for the user by Program Management. It is the typical unit of loading and execution. The program itself is a static entity, that is, it is the object files and libraries which get established and linked for each separate execution of the program. Each one of those executions is a separate task.

Each task represents a separate loading and execution environment. Any common blocks (i.e., FORTRAN common, PL/I static external, COBOL global) declared in the task are accessible by any procedure in the task. All entry point - external reference matchings with the exception of gate linkages are evaluated in the task context. No data is automatically shared between tasks in the same job, however, since they are in the same address space, sharing segments is facilitated.

2.1.3 SUBTASK

A procedure is a logically discrete piece of code that is the basic component of a program. A procedure may be compiled with other procedures to form a single object module; may be bound by the library generator with other object modules, or may be linked to other discrete object modules at execution time.

A subtask is an asynchronous activation of a procedure. A procedure whether called as a "subroutine" within a program or called asynchronously has a single allocation of data associated with in at call time. The data is the variables that are local to the procedure in the block structured language sense (e.g., PL/1 automatic). The allocation is made at call time in the run time stack segment provided by Program Management. In the case of the spawning of a subtask, a new stack segment is provided for the subtask for its stack frame and the stack frames of any procedures serially called by the subtask. This is the only private data associated with the subtask. All static data and linkages associated with the spawner are associated with the spawned subtask as well. A subtask is intended to be the most efficiently established asynchronous facility supported by Program Management. This will be effected by only providing it with the minimum necessary amount of environment.

2.2 <u>TASK ESTABLISHMENT</u>

A task is defined to Program Management with a Task Control Block (TCB). The TCB specifies the program to be executed and its execution environment. A task is established by issuing a PM#EXECUTE request. Task establishment consists of loading a program, and creating a control point with an exchange package and stacks for established programs in different rings that will be called during the course of execution. The simplest task example would be one with an exchange package, a stack for the user program, and a stack for the task services program.

Subsystem Services programs can be established and included in the execution environment. A control point and stacks are created by a PM#EXECUTE request but not by a PM#ESTABLISH request. Both effect the loading of a specified program.

2.2.1 LOADING

A program is defined to Program Management with a Program Control Block (PCB). The PCB specifies a list of object files, a list of library files, and an entry point for the program. The Loader uses this information to construct an object module segment, a working storage segment, and a binding segment.

First the Loader builds the object module segment from the list of object files, if specified. An object file is generated

by a compiler and may contain one or more object modules that represent code in a nonexecutable form. The format is detailed in Chapter 11 of the OSGDS. For each object module, the Loader creates an executable code section in the object module segment, a working storage section and a binding section.

Next the Loader resolves unsatisfied externals using the library segments. The listed library segments are represented (via SC#INITIATE_SEGMENT) in the address space of a job as is, with one process segment per library. File attachments must be done prior to this step. A library segment contains an entry point dictionary and one or more load modules. The difference between a load module and an object module is that the code section of a load module is already in executable form. For each referenced load module, the Loader generates a working storage section and a binding section in the corresponding segments.

From the list of object files, every object module, referenced or not, is loaded resulting in Loader Symbol Table entries, a working storage section, and a binding section. Only referenced load modules are loaded.

Library segments may be shared by jobs. Programs using the same object file get separate object module segments built by the Loader.

The search order used by the Loader when resolving an external reference is as follows:

- o Loader Symbol Table
- o Dictionary on each library in the order of the list.
- o Job Gate Table

2.2.2 TABLES

2.2.2.1 <u>Program Control Block</u>

The program control block (PCB) is an LNS structure used to define a program to the system. It has the following items:

- o Primary entry point - the name of the entry point at which to begin execution of the program. An alternate starting entry point can be specified in a task control block.
- o Binary object file list - the LNS name of a list of binary object files, each of which containing one or more

contiguous object modules.
- o Library list - the LNS name of a list of libraries. Each library segment contains one or more load modules and dictionaries organized by entry point name that are used to locate procedures. All load modules in a library have the same segment attributes.
- o Size - the initial working set size for the program. It is the number of page frames needed by any execution of the program when first brought into core by the Running Job Monitor.
- o Ring - the ring of execution for the program. If specified, it must be within the execution bracket for all the files and segments specified in the PCB.
- o Termination entry point - an optional field specifying an entry point name for a termination procedure. If present, the termination procedure will be called by the system during the orderly process of task termination. Parameters will be passed indicating a normal or abnormal termination.

2.2.2.2 Task Control Block

The task control block (TCB) is an LNS structure used to define the execution environment for a program. It has the following items:
- o PCB - the LNS name of the program control block that defines the program to be executed by this task.
- o Entry point - an alternate entry point name at which to begin execution of the program. If specified, would override the primary entry point as named in the program control block. The alternate entry point could change the definition of the program.
- o Size - an alternate working set size for the program being executed. If specified, would override the initial working set size in the program control block.
- o Parameters - the parameter block pointed to at entry.
- o Loader map - options indicating the level of detail to be generated for a loader map.
- o Abort - options indicating the kind of dump required on an abort.
- o Exit - the type of exit (normal, abnormal) taken by this task via the PM#EXIT request.
- o Code- an integer completion code specified on the PM#EXIT request by this task.
- o Message - a completion message up to 31 characters specified on the PM#EXIT request by this task.

- o EPCB - pointer to the Established Program Control Block for this task. Placed here by the Establisher.

2.2.2.3 Established Program Control Block

The established program control block (EPCB) is a structure internal to Program Management and is used to define the loaded environment for a program. The EPCB can be the result of either a PM#EXECUTE request or a PM#ESTABLISH request and has the following items:
- o How established - indicates established by PM#EXECUTE or by PM#ESTABLISH.
- o TCB - the LNS locator of the task control block specified on either request.
- o PCB - the LNS locator of the program control block.
- o JCB - the LNS locator of the job control block for the job in which the program is established. This field is used to obtain Job Gate Table and Job Stack Table entries for any further linking and stack allocation in this job.
- o Ring - the ring in which the program is to be executed.
- o Loader symool table - pointer to the loader symbol table for this program.
- o Binary object file list - the same list specified in the PCB but in a format more convenient for use by the loader.
- o Library list - the same list specified in the PCB but in a format more convenient for use by the loader.
- o Thread - the EPCBs are threaded together on a job basis. The starting point is in the JCB.
- o Keys - the global and local key (not supported in V 1.0).
- o Event - pointer to the event control block of the task completion event for the task as specified on the PM#EXECUTE request.
- o Control point - the control point id for the task.
- o Dependencies - task dependency threads for future use.
- o LNS search list - pointer to the LNS search list for this established program.

2.2.2.4 Job Gate Table

The Job Gate Table (JGT) is a structure internal to Program Management and is used to register gated entry points on a job basis. The JGT is searched by the Loader when resolving external

2.0 PROGRAM EXECUTION
2.2.2.4 Job Gate Table

references. An entry point is registered in the JGT during the loading of a module that possesses the gate attribute. All the entry points of such a module are registered as gates.

Gate is the mechanism used to satisfy the requirement of protecting one program from another by allowing entry to the protected code at defined points. Not only does the Loader put a gated entry point in the JGT but also marks it in the binding section so the hardware can enforce the protection. The user cannot write a binding section.

2.2.2.5 Job Stack Table

The Job Stack Table (JST) is a structure internal to Program Management and is used to allocate stacks when a control point is created. It is a job local array of integers indicating the number of stacks to allocate on a ring-by-ring basis. When a job is created, some minimum set of job and execution tables are built. The JST is included in this job template specifying stack allocation for Task Services. Using the PM#ESTABLISH request will change the JST for the specified program establishment ring.

2.2.3 TASK ESTABLISHMENT EXAMPLE

The purpose of the following example is to show structures visible to the user that make up the execution environment for his program. The example starts at the point execution is asked for via SCL, which in turn issues the request:

PM#EXECUTE (task, event, status)

task: the LNS descriptor of "USER_TCB" obtained by SCL via the LNS#ENTRY request.

event: not used in this example.

status: request status returned to SCL.

Figure 2.2-1 shows the relationship of LNS structures declared prior to issuing PM#EXECUTE. For these structures, the diagram includes only those LNS fields necessary to the example.

o "USER_TCB": local LNS name of the Task Control Block

2.0 PROGRAM EXECUTION
2.2.3 TASK ESTABLISHMENT EXAMPLE

specifying the program to execute and the parameters retrievable by that program.

o "USER_PCB": local LNS name of the Program Control Block defining the program via a list of object files and a list of library segments.

o "USER_OBJ_LIST": local LNS name of list of object files generated by prior compilation. The object modules on these files will be converted to code sections in the object module segment.

o "USER_LIB_LIST": local LNS name of the list of library segments to be used to search in the order listed for unresolved external references.

o "OBJ_FILE_1" and "OBJ_FILE_2": local LNS names of the File Control Blocks describing the object files to be loaded. The example has each file containing one module, A and B respectively.

o "USER_LIBRARY": local LNS name of a File Control Block describing the library file of the user. Only the referenced modules of this library will be loaded. The user can convert object files to library segments by using the library generator, OBLIGE.

o "COBOL_RUN_TIME": global LNS name of a File Control Block describing the library segment of COBOL run time routines. This library segment is generated by the installation and is shared by users.

Figure 2.2-2 shows the user segments created through program loading for the PM#EXECUTE request. Note that working storage sections and binding sections are created for every object module but not every load module. Only the user stack segment is shown. There would also be a stack segment allocated via the JST for Task Services.

Figure 2.2-3 shows additions to user segments resulting from the request:

PM#LOAD (name, type, pointer, status)

name: name of an entry point in load module D on the user library file.

type: type of pointer to be used in a reference to the

2.0 PROGRAM EXECUTION
2.2.3 TASK ESTABLISHMENT EXAMPLE

module.

pointer: the returned pointer after loading.

status: returned request status.

The load module D does not have any externals causing the loading of any other modules. Had it any, those load modules contained the matching entry points would have been loaded by PM#LOAD as well.

2.0 PROGRAM EXECUTION
2.2.3 TASK ESTABLISHMENT EXAMPLE

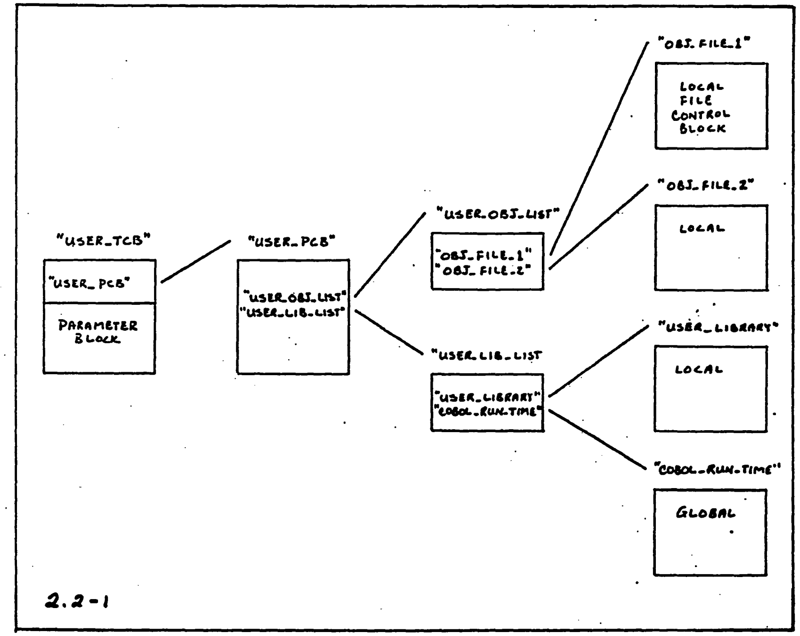

Figure 2.2-1
LNS STRUCTURES

2.0 PROGRAM EXECUTION
2.2.3 TASK ESTABLISHMENT EXAMPLE

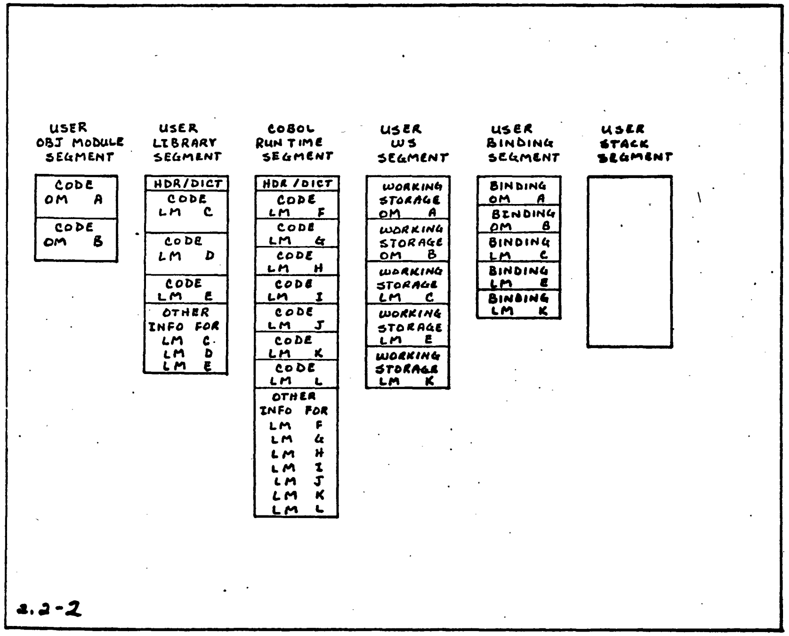

Figure 2.2-2
USER SEGMENTS

2.0 PROGRAM EXECUTION
2.2.3 TASK ESTABLISHMENT EXAMPLE

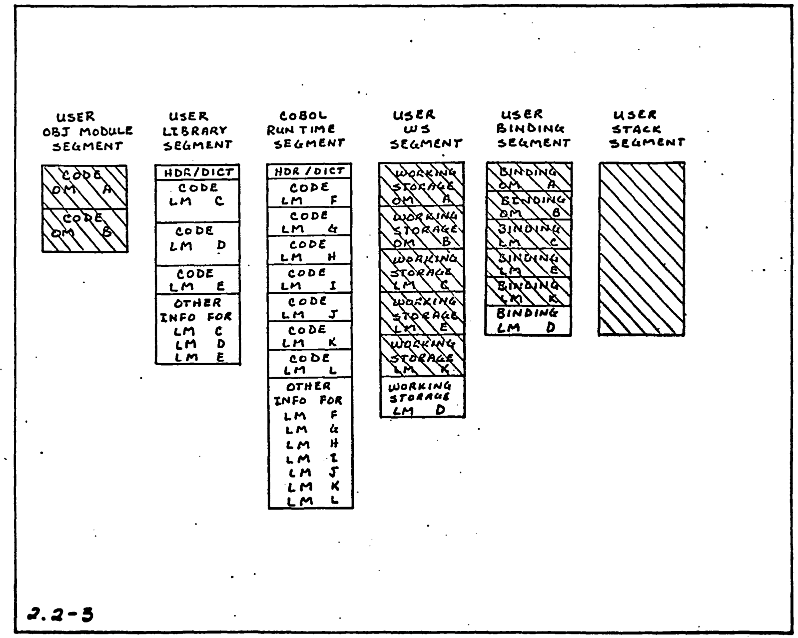

Figure 2.2-3
ADDITIONAL SEGMENT SECTIONS

2.2.4 SUBTASK ESTABLISHMENT

2.2.4.1 Subtask Control Block

The subtask control block is an LNS structure, the declaration of which causes the allocation and initialization of a control point and the allocation of stacks according to the Job Stack Table. It has the following items:

- o Control point - the control point id typically passed by Task Services in the body of a signal being sent to a System Task in the System Job. That System Task then has a return signal address to be used when sending a response.

2.3 PROGRAM EXECUTION REQUESTS

There are several levels of documentation that will eventually exist for interfacing to program execution:

- o Command Language statements
- o Control Language macros
- o Requests
- o Calls

Documentation for calls will detail three parameters in SWL:

- o Request code
- o Returned request status
- o Request block

This documentation will be provided as soon as request definitions have been compiled.

Control Language macros may not necessarily be one-to-one with the calls. There may be some calls not visible in the Control Language. Likewise, there may be some Control Language macros not externalized through the Command Language.

The Control Language macros for program execution are as follows: (To be supplied).

Request documentation is simply a prose description of a function performed and the parameters supplied by the requestor. Requests are one-to-one with calls. The program execution requests are follows:

```
PM#EXECUTE (task, event, status)
PM#EXIT (type, code, message)
PM#TERMINATE (task, status)
PM#SPAWN (entry, parameters, subtask, event, status)
PM#LOAD (name, type, pointer, status)
PM#ENTRY (name, gate, segment, type, pointer, status)
PM#REINITIALIZE (name, status)
PM#ESTABLISH (task, status)
PM#DISESTABLISH (task, status)
```

2.3.1 PM#EXECUTE

This request is used to load a program and create a task to asynchronously execute that program.

PM#EXECUTE (task, event, status)

task: the LNS descriptor of a previously declared task control block used by the requestor to identify and control task execution. The task control block identifies the program control block of the program to be loaded and executed.

event: optional parameter that is a pointer to an event control block to be associated with task completion. If specified, Program Management will cause the event when task completion is detected.

status: returned request status.

2.3.2 PM#EXIT

This request is used to indicate task completion.

PM#EXIT (type, code, message)

type: indicates the type of exit being taken, normal or abnormal. The exit_type is put in the task control block by the PM#EXIT request processor.

code: a programmer defined integer put in the task control block by the PM#EXIT request processor.

message: a programmer defined message up to 31 characters put in the task control block by the PM#EXIT request processor.

2.3.3 PM#TERMINATE

This request is used by a task to terminate another task.

PM#TERMINATE (task, status)

task: the LNS descriptor of the task control block of the task to be terminated. The TCB must be one used for a previous PM#EXECUTE request.

status: returned request status.

2.3.4 PM#SPAWN

This request is used to start an asynchronous execution of a subtask within a task.

PM#SPAWN (entry, parameters, subtask, status)

entry: pointer to procedure at which to start asynchronous execution.

parameters: pointer to argument list for the procedure.

subtask: the LNS descriptor of a previously declared subtask control block, which resulted in allocations of a control point and stacks.

status: returned request status.

2.3.5 PM#LOAD

This request is used to load a procedure not yet referenced in a program.

PM#LOAD (name, type, pointer, status)

name: entry point name.

type: the type of pointer to be returned. Can specify return of a 48 bit pointer, a code base pointer, or a code base-binding section pair.

pointer: returned pointer according to specified type wanted.

status: returned request status.

2.3.6 PM#ENTRY

This request is used to retrieve a pointer to be used in a call to a specified entry point. The module containing the entry point must have been previously loaded. The order of search for the entry point is the same for loading (a Loader Symbol Table and then the Job Gate Table).

PM#ENTRY (name, gate, segment, type, pointer, status)

name: entry point name.

gate: optional parameter indicating search should be only on the Job Gate Table.

segment: optional parameter indicating the segment which dictates the LST to start the search. Segment numbers in a job are unique per establishment of a program.

type: type of pointer to be returned. Can specify return of a 48 bit pointer, a code base pointer, or a code base-binding section pair.

pointer: returned pointer according to type.

status: returned request status.

2.3.7 PM#REINITIALIZE

The purpose of this request is to provide COBOL an Operating System function necessary to satisfy their implementation of the ANSI standard CANCEL statement. This assumes the implementation of their CALL statement would use our PM#LOAD request. We do not currently know exactly what is required of the Operating System to satisfy any requirement imposed by a CANCEL implementation.

PM#REINITIALIZE (name, status)

name: entry point name.

status: returned request status.

2.3.8 PM#ESTABLISH

This request is used to establish a program in the address space of a job. The primary purpose of the request is to establish subsystem services on a job basis.

PM#ESTABLISH (task, status)

task: the LNS descriptor of a previously declared task control block used by the requestor to identify and control the loading of a program. The task control block identifies the program control block of the program to be loaded.

status: returned request status.

2.3.9 PM#DISESTABLISH

This request is used to remove an established program from the address space of the job.

PM#DISESTABLISH (task, status)

task: the LNS descriptor of the task control block describing the program that was established.

status: returned request status.

3.0 LOGICAL NAME SPACE MANAGEMENT

This document is the GDS for the Logical Name Space manager for IPL/OS.

The functions described are the basic capabilities of the subsystem. As the OS requirements for LNS services become better defined, more sophisticated functions will be built using these basic capabilities.

3.0.1 DESIGN OBJECTIVES

The design objectives of the Logical Name Space manager are as follows.

- To provide a generalized technique for the mapping of names to data.

- To provide a symbol table handler for System Command Language.

- To apply structuring methods to dynamic OS data compatible with SWL data representation for OS code and SCL for user manipulation. (i.e. records, arrays, etc.)

- To retain certain attributes of data to allow generic requests that may operate on several types of data or resources.

- To provide a degree of data protection and privacy by a hierarcical block structure of data segments while allowing the explicit sharing of data when required.

3.1 SYSTEM DESCRIPTION

The logical name space (LNS) is composed of user and system supplied segments containing user and system defined entries. A list called the LNS segment list is maintained for each job known to the system. The LNS segment list contains the names of the segments which are to be searched for LNS entries and the order in which they are to be searched. When an LNS entry is sought each segment whose name appears in the LNS segment list is searched until the entry has been found or the list has been exhausted. The segment whose name appears in the last slot of the LNS segment list is called the most local segment and is searched first.

All internal LNS information uses relocatable addressing enabling a segment to be established at any virtual address while preserving previously defined information.

During job initiation the system allocates an LNS segment list and initializes it as follows.

LNS#GLOBAL system global segment

.

```
   .                        other segments
   .
   LNS#LOCAL                most local segment
```

Each entry has an internal entry descriptor. These internal descriptors are managed in several chains per segment with a name hashing algorithm randomly assigning an entry to a chain. The entry search strategy includes a percolating of the internal descriptor chain which results in the chain being ordered by most recent use. Item chains are handled in the same manner.

An entry and its internal descriptor form the primary node of a data structure through which the user can descend to any level.

3.1.1 LNS DESCRIPTORS

Each entry or item in the LNS has an internal LNS descriptor associated with it which is NOT accessable to the user. The definition of this internal descriptor is as follows.

```
type_desc = RECORD
   desc_type: (entry, item),  "type of descriptor"
   lock: BOOLEAN,  "internal synchronization lock"
   chain: REL ^type_desc,  "chain to next descriptor"
   name: STRING (31) OF CHAR,  "name of entry or item"
   hash: 0..255,  "hash value of name"
   excl: STRING (31) OF CHAR,  "exclusive lock key"
   non_excl: 0..65565,  "non-exclusive lock count"
   data_type: 0..max_type,  "subscript to LNS#TYPE table"
   data_len: 0..max_len,  "string or set length"
   data_dim: 0..max_dim,  "dimension of array variable"
   data: REL ^type_data,  "location of data"
   ex_attr: SET OF 1..64,  "extrinsic attributes"
   RECEND,
```

A complex type is described by an array of internal field descriptors. This array exists only once in the global segment regardless of the number of occurences of the complex type. The definition of the internal field descriptor is as follows.

```
type_field_desc = ARRAY [*] OF RECORD
   name: STRING (31) of CHAR,  "name of field"
   hash: 0..255,  "hash value of name"
   data_type: 0..max_type,  "subscript to LNS#TYPE table"
   data_len: 0..max_len,  "string or set length"
   data_dim: 0..max_dim,  "dimension of array variable"
   data: REL ^type_data,  "location of field in record"
   ex_attr: SET OF 1..64,  "extrinsic attributes"
   RECEND,
```

Several of the LNS requests require or return a descriptor. This descriptor resides in the users memory and is fully accessable. The definition of this descriptor is as follows.

```
type_user_desc = RECORD
   data_type: 0..max_type,  "subscript to LNS#TYPE table"
   data_len: 0..max_len,  "string or set length"
   data_dim: 0..max_dim,  "dimension of array variable"
   data_size: 0..max_size,  "size of data in cells"
   excl: BOOLEAN,  "exclusive lock on"
   non_excl: BOOLEAN,  "non-exclusive lock(s) on"
   data: ^type_data,  "location of data"
   desc: ^type_desc,  "location of internal descriptor"
   ex_attr: SET OF 1..64,  "extrinsic attributes"
   RECEND,
```


3.1.2 LNS DATA TYPES

LNS data types fall into two classes; simple and complex. A simple type or a complex type may be an element of a complex type. The currently defined simple types are as follows.

UNKNOWN	undefined type (undeclarable)
INTEGER	integer variable
REAL	real variable
BOOLEAN	boolean variable
STRING	character string variable
SET	set variable
POINTER	pointer variable
CELL	cell variable
ALIAS	LNS alias name
CHAIN	LNS item chain

The following additional subsets of INTEGER will be included when their IPL/SWL representations are defined.

SUBRANGE	i..j
ORDINAL	(x, y, z, p, LNS)

The following SWL representation attributes are not supported at this stage of the LNS design.

PACKED
CRAMMED

The currently defined complex types are as follows.

SCL#TOKEN	SCL token
SCL#OPERATOR	SCL operator
SCL#FUNCTION	SCL function
SCL#COMMAND	SCL command
SCL#MACRO	SCL macro

3.1.3 LNS STRUCTURES

By the use of cataloged internal descriptors of complex types and chained items, arbitrarily complex structures may be assembled.

3.1.3.1 General Examples

Examples of the commands to build the structures shown are included using Command Language syntax for clarity. The following symbols are used in the diagrams of these structures.

e	entry name
f	field name
i	item name
d	LNS internal descriptor
c	chain linkage
x	data

The following structure is built for scalar numeric variables.

```
eeeeeeee
dddddddd...
         :
         :.>xxxxxxxx
```

Example:
lns#declare name,integer

The following structure is built for numeric arrays.

```
eeeeeeee
dddddddd...
         :
         :.>xxxxxxxx
            xxxxxxxx
            xxxxxxxx
            xxxxxxxx
```

Example:
lns#declare name,real,dim=4

The following structure is built for scalar string variables.

```
eeeeeeee
dddddddd...
         :
         :.>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Example:
lns#declare name,string

The following structure is built for string arrays.

```
eeeeeeee
dddddddd...
         :
         :.>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
            xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Example:
lns#declare name,string,32,4

3.0 LOGICAL NAME SPACE MANAGEMENT
3.1.3.1 General Examples

The following structure is built for complex types.

```
eeeeeeee
dddddddd.................>ffffffff
          :            ...dddddddd
          :.>xxxxxxxx<.:  ffffffff
             xxxxxxxx<....dddddddd
          ...cccccccc<..  ffffffff
          :            :..dddddddd
          :
          :.>iiiiiiii  ..>iiiiiiii  ..>iiiiiiii
             cccccccc..:  cccccccc..:  cccccccc
          ...dddddddd  ...dddddddd  ...dddddddd
          :            :            :
          :.>xxxxxxxx  :.>xxxxxxxx  :.>xxxxxxxx
```

Example:

```
Ins#record record_name,3
Ins#field record_name,field_a,integer
Ins#field record_name,field_b,real
Ins#field record_name,field_c,chain
Ins#declare structure,type=record_name
Ins#insert structure.field_c,item_a
Ins#insert structure.field_c,item_b
Ins#insert structure.field_c,item_c
```


3.0 LOGICAL NAME SPACE MANAGEMENT
3.1.3.1 General Examples

As illustrated below, all combinations of items and fields are permitted.

```
eeeeeeee
dddddddd.................>ffffffff
          :            ...dddddddd
          :.>xxxxxxxx<.:  ffffffff
             xxxxxxxx<....dddddddd
          ...cccccccc<... ffffffff
          :  xxxxxxxx<. :.dddddddd
          :            :  ffffffff
          :            :...dddddddd
          :
          :.>iiiiiiii  ..>iiiiiiii  ..>iiiiiiii
             cccccccc..:  cccccccc..:  cccccccc
          ...dddddddd  ...dddddddd  ...dddddddd
          :            :            :
          :            :.>xxxxxxxx  :.>cccccccc...
          :                                   :
          :..............>ffffffff      iiiiiiii<.:
          :            ...dddddddd      cccccccc
          :.>xxxxxxxx<.:  ffffffff      dddddddd...
             xxxxxxxx<....dddddddd...             :
             xxxxxxxx<. :           :  xxxxxxxx<.:
             xxxxxxxx  : : ffffffff<.:
                       : :.dddddddd
                       :   ffffffff
                       :...dddddddd
```

Example:

```
Ins#record subrecord_b,2
Ins#field subrecord_b,field_a,real
Ins#field subrecord_b,field_b,dim=2
Ins#record suprecord_a,2
Ins#field subrecord_a,field_a,integer
Ins#field suprecord_a,field_b,type=subrecord_b
Ins#record record_name,4
Ins#field record_name,field_a,integer
Ins#field record_name,field_b,real
Ins#field record_name,field_c,chain
Ins#field record_name,field_d,string,8
Ins#declare name,type=record_name
Ins#insert name.field_c,item_a,type=subrecord_a
Ins#insert name.field_c,item_b
Ins#insert name.field_c,item_c,type=chain
Ins#insert name.field_c.item_c,item_a,real
```

3.1.3.2 SCL#TOKEN Example

SWL Definition

```
 TYPE

    lns#desc = RECORD
       data_type: 0..max_type,
       data_len: 0..max_len,
       data_dim: 0..max_dim,
       data_size: 0..max_size,
       excl: BOOLEAN,
       non_excl: BOOLEAN,
       data: ^type_data,
       desc: ^type_desc,
       ex_attr: SET OF 1..64,
       RECEND,

    scl#string = RECORD,
       lhi: 1..256,
       rhi: 0..255,
       buff: STRING (255) OF CHAR,
       RECEND,

    scl#token = RECORD
       typ: INTEGER,
       desc: lns#desc,
       iv: INTEGER,
       rv: REAL,
       sv: scl#string,
       RECEND;
```

LNS Definition

```
    lns#record lns#desc,9,(declare,insert),lns#103
       lns#field lns#desc,data_type
       lns#field lns#desc,data_len
       lns#field lns#desc,data_dim
       lns#field lns#desc,data_size
       lns#field lns#desc,excl,boolean
       lns#field lns#desc,non_excl,boolean
       lns#field lns#desc,data,pointer
       lns#field lns#desc,desc,pointer
       lns#field lns#desc,ex_attr,set,64

    lns#record scl#string,3,(declare,insert),lns#103
```



```
       lns#field scl#string,lhi
       lns#field scl#string,rhi
       lns#field scl#string,buff,string,255

    lns#record scl#token,5
       lns#field scl#token,typ
       lns#field scl#token,desc,lns#desc
       lns#field scl#token,iv
       lns#field scl#token,rv,real
       lns#field scl#token,sv,scl#string
```

3.1.4 TYPE CONTROLLED "OWN CODE" PROCEDURES

Provision has been made for "own code" procedures external to the LNS system code to be conditionally called by request and data type. This feature is intended to allow other components of the operating system to be advised of certain LNS operations on data with which they are concerned. These procedures will be passed the LNS request code, the request parameters and the LNS internal descriptor of the entry or item to be acted upon. A status must be returned by the procedure. If this status is not normal, the LNS request will be aborted and the status returned to the user. All parameters and status records will follow normal system conventions.

These procedures are defined at the time the internal field descriptor is catalogued by the LNS#RECORD request. The dynamic loader is called with the named procedure to supply a value for a procedure pointer variable (^PROC) in the LNS#TYPE table. This procedure is called by the specified LNS requests whenever a variable or a field of a variable of the specified type is referenced. The trap procedure is invoked prior to any action by the LNS request with the exception of LNS#DECLARE and LNS#INSERT. These two requests call the procedure after completing their respective functions. A trap procedure may issue LNS requests. However, recursion and interlock problems are possible if the logic of the trap is defective.

Two examples of possible uses of these procedures are to initialize variables when declared or inserted or to monitor changes via get and put by the user to tables currently in use by the system. A third example would be the implicit declaration of associated variables or insertion of items into a chain field.

A set of procedures named in the form LNS#<status code> are supplied in the LNS library. These procedures function as own code trap routines and return the status record indicated by their name. For example LNS#103 returns a status of "invalid type" and may be used to prevent declaration of a complex type intended only for use as a field of another complex type and never as a variable on its own. This is illustrated in the SCL#TOKEN example.

3.1.5 EXTRINSIC ATTRIBUTES

In addition to the normal LNS intrinsic attributes, the LNS system will retain 64 user defined extrinsic attributes for any element. These attributes have no meaning to the LNS system but may be assigned and queried by the user.

Attributes 33 to 64 are reserved for operating system use (i.e. SCL decoding attributes) while attributes 1 to 32 may be manipulated by the end user (i.e. problem program).

Currently reserved attributes are:

33..40	system command language

3.2 LNS REQUESTS

The following requests are available for the manipulation of LNS.

3.2.1 LNS#ATTACH

The purpose of the LNS#ATTACH request is to add a new segment to the LNS segment list as the most local segment.

LNS#ATTACH (segment, old, status)

segment: The segment parameter specifies a string containing the name of a segment currently known to the job (i.e. mapped in).

old: The old parameter specifies a boolean variable. If the value of the variable is true, the LNS data currently in the segment will be accessable. If the variable is false, the segment will be initialized as empty.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.2 LNS#DETACH

The purpose of the LNS#DETACH request is to remove a segment from the LNS segment list.

LNS#DETACH (segment, status)

segment: The segment parameter specifies a string containing the name of the segment to be detached. Omission of this parameter (indicated by a blank string) will cause the most local segment to be detached.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.3 LNS#DECLARE

The purpose of the LNS#DECLARE request is to declare an entry in the LNS.

LNS#DECLARE (segment, entry, type, length, dim, status)

segment: The segment parameter specifies a string containing the name of the segment in which the entry is to be declared. Omission of the segment parameter (indicated by a blank string) will cause the entry to be declared in the most local segment.

entry: The entry parameter specifies a string containing the name of the entry being declared.

type: The type parameter specifies a string containing the type of the entry being declared. Omission of the type parameter (indicated by a blank string) will cause an entry of type INTEGER to be declared. The valid LNS types are those described under "data types" or any complex type previously defined by LNS#RECORD and LNS#FIELD.

length: The length parameter is only meaningful when declaring string or set variables. For strings the parameter specifies an integer containing the number of bytes to be allocated for the string. For set variables the integer contains the number of elements in the set. Omission of the length parameter (indicated by a 0) will cause a default of 32 to be assumed.

dim: The dim parameter specifies an integer containing the dimension of the entry being declared. Omission of the dim parameter (indicated by a 0) will cause a default of 1 to be assumed.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.4 LNS#REMOVE

The purpose of the LNS#REMOVE request is to remove an entry from the LNS.

LNS#REMOVE (segment, entry, status)

segment: The segment parameter specifies a string containing the name of the segment which is to be searched for the entry. Omission of the segment parameter (indicated by a blank string) will cause each segment whose name appears in the LNS segment list to be searched.

entry: The entry parameter specifies a string containing the name of the entry which is to be deleted.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.5 LNS#ENTRY

The purpose of the LNS#ENTRY request is to get the descriptor of an LNS entry given the name of the entry.

LNS#ENTRY (segment, entry, subscr, desc, status)

segment: The segment parameter specifies a string containing the name of the segment which is to be searched for the entry. Omission of the segment parameter (indicated by a blank string) will cause each segment whose name appears in the LNS segment list to be searched.

entry: The entry parameter specifies a string containing the name of the entry whose descriptor is being sought.

subscr: The subscr parameter specifies an integer containing the subscript to be used when the entry is an array. Omission of the subscr parameter (indicated by a 0) will cause a descriptor of the entire array to be returned.

desc: The desc parameter specifies a record into which a descriptor is to be returned.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.6 LNS#NEXT

The purpose of the LNS#NEXT request is to get the descriptor of a field or item given the descriptor of the enclosing entry, item or field.

LNS#NEXT (input_desc, name, subscr, output_desc, status)

input_desc: The input_desc parameter specifies the name of a record containing a descriptor of the enclosing entry, field or item.

name: The name parameter specifies a string containing the name of the field or item whose descriptor is being sought.

subscr: The subscr parameter specifies an integer containing the subscript to be used when the field or item is an array. Omission of the subscr parameter (indicated by a 0) will cause a descriptor of the entire array to be returned.

output_desc: The output_desc parameter specifies a record into which the descriptor of the field is to be placed.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.7 LNS#SLICE

The purpose of the LNS#SLICE request is to get the descriptor of an element of an array given the descriptor of the array.

LNS#SLICE (input_desc, subscr, output_desc, status)

input_desc: The input_desc parameter specifies a record containing the descriptor of the array to be sliced.

subscr: The subscr parameter specifies an integer containing the subscript of the desired element.

output_desc: The output_desc parameter specifies a record into which the descriptor of the element is to be placed.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.8 LNS#GROW

The purpose of the LNS#GROW request is to grow the dimension of an LNS entry or item.

LNS#GROW (desc, incr, status)

desc: The desc parameter specifies a record containing a descriptor of the entry or item whose dimension is to be grown.

incr: The incr parameter specifies an integer containing the increment by which the dimension of the entry is to be grown. Omission of the incr parameter (indicated by a 0) will cause a default of 1 to be assumed.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.9 LNS#LOCK

The purpose of the LNS#LOCK request is to lock an LNS entry or item. The locking operation has no effect on other requests except other LNS#LOCKs, LNS#UNLOCK, LNS#REMOVE and LNS#DELETE.

LNS#LOCK (desc, excl, key, status)

desc: The desc parameter specifies a record containing a descriptor of the entry or item to be locked.

excl: The excl parameter specifies a boolean variable. If the value of the variable is true, access to the entry will be exclusive. If the value is false, access will be non-exclusive.

key: The key parameter specifies a string of 31 characters in which a unique name will be returned when the access requested was exclusive. If non-exclusive access was requested, the contents are unchanged.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.10 LNS#UNLOCK

The purpose of the LNS#UNLOCK request is to unlock an LNS entry or item.

LNS#UNLOCK (desc, key, status)

desc: The desc parameter specifies a record containing a descriptor of the entry or item to be unlocked.

key: The key parameter specifies a string containing the string returned by the LNS#LOCK request when the entry was locked for exclusive access. If the entry is being unlocked from non-exclusive access the key parameter is ignored.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.11 LNS#INSERT

The purpose of the LNS#INSERT request is to insert a new item into a chain.

LNS#INSERT (desc, item, type, length, dim, status)

desc: The desc parameter specifies a record containing a descriptor of the entry, field or item within which the item is to be allocated.

item: The item parameter specifies a string containing the name of the item to be allocated.

type: The type parameter specifies a string containing the type of the item being inserted. Omission of the type parameter (indicated by a blank string) will cause an item of type INTEGER to be inserted. The valid LNS types are those described under "data types" or any complex type previously defined by LNS#RECORD and LNS#FIELD.

length: The length parameter is only meaningful when inserting string or set items. For strings the parameter specifies an integer containing the number of bytes to be allocated for the string. For set items the integer contains the number of elements in the set. Omission of the length parameter (indicated by a 0) will cause a default of 32 to be assumed.

dim: The dim parameter specifies an integer containing the dimension of the item being inserted. Omission of the dim parameter (indicated by a 0) will cause a default of 1 to be assumed.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

NOTE: The trap to an "own code" procedure by the LNS#INSERT request is determined by the data type of the item being inserted.

3.2.12 LNS#DELETE

The purpose of the LNS#DELETE request is to delete an item from a chain.

LNS#DELETE (desc, item, status)

desc: The desc parameter specifies a record containing a descriptor of the entry, field or item which is to be searched for the item.

item: The item parameter specifies a string containing the name of the item to be deleted.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

NOTE: The trap to an "own code" procedure by the LNS#DELETE request is determined by the data type of the item being deleted.

3.2.13 LNS#GET

The purpose of the LNS#GET request is to get a value from the LNS.

LNS#GET (desc, buffer, status)

desc: The desc parameter specifies a record containing a descriptor of the entry, field or item whose value is being sought.

buffer: The buffer parameter specifies a buffer into which the value is to be placed.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.14 LNS#PUT

The purpose of the LNS#PUT request is to put a value into the LNS.

LNS#PUT (desc, buffer, status)

desc: The desc parameter specifies a record containing a descriptor of the entry, field or item whose value is to be updated.

buffer: The buffer parameter specifies the buffer containing the new value.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.2.15 LNS#SETXA

The purpose of the LNS#SETXA request is to set the extrinsic attributes of an entry or item. Permission to alter attributes 33..64 is verified by the OS#CHECK procedure.

LNS#SETXA (desc, attr, status)

desc: The desc parameter specifies a record containing a descriptor of the entry or item whose extrinsic attributes are to be set.

attr: The attr parameter specifies a set of 1..64 containing the attributes to be changed. This set will be "xored" to the current set of attributes resulting in the symetric difference of the two sets. In other words, the presence of any attribute in this parameter causes the attribute to be "toggled" in the LNS internal descriptor.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.3 PRIVILEGED REQUESTS

The following requests are subject to restrictions such as Operating System only or SE#OP use. When any of these requests are issued permission is verified by the OS#CHECK procedure.

3.3.1 LNS#RECORD

The purpose of the LNS#RECORD request is to define a new complex type to the system. The type definition is always global.

LNS#RECORD (record, fields, traps, procedure, status)

record: The record parameter specifies a string containing the name of the complex type to be defined.

fields: The fields parameter specifies an integer containing the maximum number of fields to exist in the complex type.

traps: The traps parameter specifies an ordered set of requests for which an "own code" procedure is to be invoked for this type. If this parameter is omitted (indicated by an empty set) no traps will occur. The positional significance of each request in the set is as follows.

```
LNS#DECLARE...............1
LNS#REMOVE................2
LNS#ENTRY.................3
LNS#NEXT..................4
LNS#SLICE.................5
LNS#GROW..................6
LNS#LOCK..................7
LNS#UNLOCK................8
LNS#INSERT................9
LNS#DELETE...............10
LNS#GET..................11
LNS#PUT..................12
LNS#SETXA................13
```

procedure: The procedure parameter specifies a string containing the name of the "own code" procedure to be invoked as indicated by the traps parameter. The procedure named must reside in a library currently known to the job. An error in this parameter will result in a status being returned from the loader rather than from LNS. If this parameter is omitted (indicated by a blank string) no traps will occur.

status: The status parameter specifies a variable into which

the status record is to be placed. The status codes returned are described under "error conditions".

3.3.2 LNS#FIELD

The purpose of the LNS#FIELD request is to define a field of a previously defined complex type.

LNS#FIELD (record, field, type, len, dim, attr, status)

record: The record parameter specifies a string containing the name of the complex type of which this field is to be a member.

field: The field parameter specifies a string containing the name of the field to be defined. This name will become the name of the first currently undefined field of the complex type.

type: The type parameter specifies a string containing the type of the field to be defined. Omission of the type parameter (indicated by a blank string) will cause a field of type INTEGER to be defined. The valid LNS types are those described under "data types" or any complex type previously defined by LNS#RECORD and LNS#FIELD.

length: The length parameter is only meaningful when defining string or set fields. For strings the parameter specifies an integer containing the number of bytes to be allocated for the string. For set fields the integer contains the number of elements in the set. Omission of the length parameter (indicated by a 0) will cause a default of 32 to be assumed.

dim: The dim parameter specifies an integer containing the dimension of the field being defined. Omission of the dim parameter (indicated by a 0) will cause a default of 1 to be assumed.

attr: The attr parameter specifies a set of 1..64 containing the extrinsic attributes to be associated with the field. Note that the LNS#SETXA request may not be used on field descriptors.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.3.3 LNS#SEGLOCK

The purpose of the LNS#SEGLOCK request is to perform a non-exclusive lock on a segment in order to prevent an LNS#DETACH request from being performed.

LNS#SEGLOCK (segment, status)

segment: The segment parameter specifies a string containing the name of the LNS segment to be locked.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.3.4 LNS#SEGUNLOCK

The purpose of the LNS#SEGUNLOCK request is to unlock an LNS segment, allowing an LNS#DETACH to be performed.

LNS#SEGUNLOCK (segment, status)

segment: The segment parameter specifies a string containing the name of the LNS segment to be unlocked.

status: The status parameter specifies a variable into which the status record is to be placed. The status codes returned are described under "error conditions".

3.4 ERROR CONDITIONS

Error conditions are represented by status information returned by each LNS request. The status information may be passed to the system message generator for further expansion and logging.

3.4.1 DEFINITION OF CODES

0	LN	000	normal completion

Parameter errors

8	LN	101	invalid segment name
8	LN	102	invalid element name
8	LN	103	invalid type
8	LN	104	invalid length
8	LN	105	invalid dimension
8	LN	106	invalid increment
8	LN	108	invalid key
8	LN	109	invalid subscript
8	LN	10A	invalid descriptor

Access errors

8	LN	201	denied access
8	LN	202	segment exists
8	LN	203	segment does not exist
8	LN	204	entry exists
8	LN	205	entry does not exist
8	LN	206	field exists
8	LN	207	field does not exist
8	LN	208	item exists
8	LN	209	item does not exist

Functional errors

8	LN	301	entry already locked
8	LN	302	entry not locked
8	LN	303	segment locked by system
8	LN	304	element not a chain
8	LN	305	element not a structure
8	LN	306	element too large
8	LN	307	segment not locked

Internal errors

C	LN	901	no memory for LNS internal descriptor
C	LN	902	no memory for data
C	LN	903	maximum number of fields exceeded
C	LN	904	maximum number of segments exceeded
C	LN	905	maximum number of types exceeded
E	LN	EEE	feature not yet supported
F	LN	FFF	disaster

3.0 LOGICAL NAME SPACE MANAGEMENT
3.4.2 ERROR CODES BY REQUEST

3.4.2 ERROR CODES BY REQUEST

	ATTACH	DETACH	DECLARE	REMOVE	ENTRY	NEXT	SLICE	GROW	LOCK	UNLOCK
0 LN 000	X	X	X	X	X	X	X	X	X	X
8 LN 101	X									
8 LN 102			X	X	X	X				
8 LN 103			X					X		
8 LN 104			X							
8 LN 105			X							
8 LN 106								X		
8 LN 108										X
8 LN 109					X	X	X			
8 LN 10A						X	X	X	X	X
8 LN 201				X						
8 LN 202	X									
8 LN 203			X	X	X					
8 LN 204			X							
8 LN 205				X	X					
8 LN 206										
8 LN 207						X				
8 LN 208										
8 LN 209						X				
8 LN 301									X	
8 LN 302										X
8 LN 303		X								
8 LN 304										
8 LN 305										
8 LN 306			X					X		
8 LN 307										
C LN 901			X							
C LN 902			X					X		
C LN 903										
C LN 904	X									
C LN 905										
E LN EEE										
F LN FFF	X	X	X	X	X	X	X	X	X	X

3.0 LOGICAL NAME SPACE MANAGEMENT
3.4.2 ERROR CODES BY REQUEST

	INSERT	DELETE	GET	PUT	SETXA	RECORD	FIELD	SEGLOCK	SEGUNLOCK
0 LN 000	X	X	X	X	X	X	X	X	X
8 LN 101								X	X
8 LN 102	X	X				X	X		
8 LN 103	X						X		
8 LN 104	X						X		
8 LN 105	X					X	X		
8 LN 106									
8 LN 108									
8 LN 109									
8 LN 10A	X	X	X	X	X				
8 LN 201									
8 LN 202									
8 LN 203								X	X
8 LN 204						X			
8 LN 205									
8 LN 206							X		
8 LN 207									
8 LN 208	X								
8 LN 209		X							
8 LN 301									
8 LN 302									
8 LN 303									
8 LN 304	X	X							
8 LN 305									
8 LN 306	X		X	X		X			
8 LN 307									X
C LN 901	X					X			
C LN 902	X					X			
C LN 903							X		
C LN 904									
C LN 905						X			
E LN EEE									
F LN FFF	X	X	X	X	X	X	X	X	X

4.0 PROGRAM COMMUNICATIONS

4.1 EVENTS

Events are system supported facilities which permit synchronization and interrupt control for asynchronous activities within a Job. An event is represented by an event control block in storage and several system requests to manipulate the control block. An event control block may be either an LNS variable or a structure in the job data base (internal static, stack, etc.).

The event control block contains the following information:
- o Condition state (caused, cleared)
- o Action state (enabled, disabled)
- o Action (attached procedure, waited)

The condition state indicates the current condition of the event, caused or cleared. The action state, enabled or disabled, directs the system to either immediately effect the specified action or delay the action. The action can be the invoking of an attached procedure, or it can be continuing the execution of an asynchronous activity in the Job that has been waiting for the causation of this event. The action may also be a combination of both for one or several control points in a Job.

Regardless of the action state being either 'enabled' or 'disabled', the system performs the specified action only when the condition state of the event changes from 'cleared' to 'caused'. There are two requests that do this change, PM#CAUSE_EVENT and PM#CAUSE_CLEAR_EVENT. The PM#CAUSE_EVENT request sets the condition state to caused and leaves it that way. The PM#CAUSE_CLEAR_EVENT request is used for pulsing. If the condition state of an event is 'caused' when either of these requests is issued, the system will not perform the specified action. The PM#CLEAR_EVENT request sets the condition state of an event to 'cleared'.

An interrupt procedure can be attached to an event by using the PM#ATTACH_PROCEDURE request. When an event to which an interrupt procedure has been attached does occur, the result will be the serial invocation of the attached procedure using the same

control point as the requestor of the PM#ATTACH_PROCEDURE request.

If the action state of the event to which the procedure is attached is 'disabled', the invocation of the procedure will be delayed until the event is enabled. Event occurrence processing for a particular event remains disabled until enabled by the PM#ENABLE_EVENT request.

The PM#DISABLE_EVENT request prevents the invocation of any and all procedures attached to a particular event. A procedure can be attached to more than one event. More than one procedure can be attached to one event.

4.1.1 EVENT REQUESTS

The event requests provided by Program Management are as follows:

PM#ATTACH_PROCEDURE (procedure, event, status)
PM#CAUSE_EVENT (event, status)
PM#CAUSE_CLEAR_EVENT (event, status)
PM#CLEAR_EVENT (event, status)
PM#DETACH_PROCEDURE (procedure, event, status)
PM#DISABLE_EVENT (event1, ..., eventM, status)
PM#ENABLE_EVENT (event1, ..., eventM, status)
PM#STATUS_EVENT (event, condition_state, action_state, waited, attached_interrupt_procedure, status)
PM#WAIT_EVENT (event1, ..., eventM, position, status)
PM#WAIT_CLEAR_EVENT (event1, ..., eventM, position, status)

4.1.1.1 PM#ATTACH_PROCEDURE

This request establishes an association of an interrupt procedure with an event.

PM#ATTACH_PROCEDURE (procedure, event, status)

procedure: pointer to the procedure to be invoked when the event occurs.

event: pointer to event control block.

status: returned request status.

4.1.1.2 PM#CAUSE_EVENT

This request sets the specified event to caused. If the event is in the cleared state when this request is made, the system performs the action, if any, as specified in the event control block. If the event is in the caused state already when this request is made, the system does not perform any specified action and informs the requestor via the returned request status. Performing the action includes the execution of all attached procedures.

PM#CAUSE_EVENT (event, status)

event: pointer to event control block.

status: returned request status.

4.1.1.3 PM#CAUSE_CLEAR_EVENT

This requests performs the action, if any, as specified in the event control block and returns to the requestor with the event in the cleared state. If the event is in the caused state already when this request is made, the system does not perform any specified action and informs the requestor via the returned requests status. Performing the action includes the execution of all attached procedures.

PM#CAUSE_CLEAR_EVENT (event, status)

event: pointer to event control block.

status: returned request status.

4.1.1.4 PM#CLEAR_EVENT

This request sets the condition state of an event to cleared.

PM#CLEAR_EVENT (event, status)

event: pointer to an event control block.

status: returned request status.

4.1.1.5 PM#DETACH_PROCEDURE

This request removes the association of an interrupt procedure with an event. The requestor must be the same as the PM#ATTACH requestor.

PM#DETACH_PROCEDURE (procedure, event, status)

procedure: pointer to procedure to no longer be associated with the specified event control block.

event: pointer to event control block.

status: returned request status.

4.1.1.6 PM#DISABLE_EVENT

This request disables event occurrence processing for an event or events. It sets the action state of a specified event to disabled.

PM#DISABLE_EVENT (event1, ..., eventM, status)

event: pointer to an event control block.

status: returned request status.

4.1.1.7 PM#ENABLE_EVENT

This request enables event occurrence processing for an event or events. It sets the action of a specified event to enabled.

PM#ENABLE_EVENT (event1, ..., eventM, status)

event: pointer to an event control block.

status: returned request status.

4.1.1.8 PM#STATUS_EVENT

This request returns the status of an event.

PM#STATUS_EVENT (event, condition_state, action_state, waited, attached_interrupt_procedure, status)

event: pointer to event control block.

condition_state: returned state indicating caused or cleared.

action_state: returned state indicating enabled or disabled.

waited: returned indication if there are any control points in the Job waiting for this event to be caused.

attached_interrupt_procedure: returned indication if there are any interrupt procedures that are attached to this event.

status: returned request status.

4.1.1.9 PM#WAIT_EVENT

This request suspends the execution of a control point until one or all of a specified number of events has occurred.

PM#WAIT_EVENT (event1, ..., eventM, position, status)

event: pointer to an event control block.

position: if specified, one event occurrence will satisfy the wait and its position (1-M) will be returned. If not specified, all the events must occur to satisfy the wait.

status: returned request status.

The system will default a time limit so a control point will not remain suspended waiting for something that will not occur. Elapsed default time limit will be reflected in the returned request status.

While a control point is suspended waiting on an event, other events can occur. These are saved until the wait is satisfied. Then they are processed in the order of their occurrence including the event or events that satisfied the wait. Processing event occurrences includes invoking any attached interrupt procedures.

The PM#WAIT_EVENT request processor does not alter the condition of an event before returning to the requestor.

More than one control point in a Job may wait on an event, in which case all are suspended until the condition state of the specified event is caused.

If a task is suspended waiting on an event and there occurs another event to which an inner-ring interrupt procedure has been attached, the system will allow the control point to execute the interrupt procedure and then be suspended again.

4.1.1.10 PM#WAIT_CLEAR_EVENT

This request suspends the execution of a control point until one or all of a specified number of events has occurred. It is the same as the PM#WAIT_EVENT request except that it returns to the requestor with the condition state of the wait satisfying event or events as cleared.

PM#WAIT_CLEAR_EVENT (event1, ..., eventM, position, status)

event: pointer to event control block.

position: if specified, one event occurrence will satisfy the wait and its position (1-M) will be returned. If not specified, all the events must occur to satisfy the wait.

status: returned request status.

The system will default a time limit so a control point will not remain suspended waiting for something that will not occur. Elapsed default time limit will be reflected in the returned request status.

If a control point is suspended waiting on an event and there occurs another event to which an inner-ring interrupt subprogram has been attached, the system will allow the control point to execute that interrupt subprogram and then be suspended again.

While a control point is suspended waiting on an event, other events can occur. These are saved until the wait is satisfied. Then they are processed in the order of their occurrence including the event or events that satisfied the wait. Processing event occurrences includes invoking any attached interrupt procedures.

More than one control point may wait on an event, in which case all are suspended until the condition state specified event is caused.

4.2 SIGNALS

Signals are short messages that are used for inter-job communications usually in the form of requests and responses. For example, system code in a User Job can send a signal to the System Job to request some specific service. The body of the signal would contain the request. It may also contain the identification of an associated event control block for an event to be caused when a response is received.

A signal may be associated with a queue via the PM#SELECT_SIGNAL request. In this case Program Management would:

1) put the signal on a queue using the PM#ENQUEUE request

2) cause the queue-associated event, if any, as noted in the queue control block.

The signal can be removed from the queue by using the PM#DEQUEUE request.

When a signal is received by the destination control point, control first goes to a general signal handler and then is routed to signal-own-code based on the type of signal. For example, say I/O is a type of signal. Then every I/O signal received by a job could be processed by an I/O signal module to do whatever is particular for an I/O signal.

The types of signals and the information contained in a signal are detailed in Chapter 9 of OSGDS.

4.2.1 SIGNAL SELECTION

The Signal Selection List (SSL) is a structure internal to Program Management and is used to register signal selections on a control point basis. The PM#SELECT_SIGNAL request associates a signal with a queue by creating an entry in the SSL. The Task Monitor Signal Handler uses the SSL to queue signals. The PM#DESELECT_SIGNAL request removes an entry from the SSL.

4.2.2 SIGNAL REQUESTS

The signal requests provided by Program Management are as follows:

PM#SEND_SIGNAL (signal, status)
PM#SELECT_SIGNAL (name, queue, status)
PM#DESELECT_SIGNAL (name, status)
PM#STATUS_SIGNAL (signal, queue, status)
PM#DISABLE_SIGNALS (status)
PM#ENABLE_SIGNALS (status)

4.2.2.1 <u>PM#SEND_SIGNAL</u>

This request sends a signal from one job to another job.

PM#SEND_SIGNAL (signal, status)

signal: pointer to the signal to be sent.

status: returned request status.

4.2.2.2 <u>PM#SELECT_SIGNAL</u>

This request associates a signal with a queue. More than one signal may be associated with one queue, but not multiple queues for a signal.

PM#SELECT_SIGNAL (name, queue, status)

name: the signal type and id in the header of the signal expected to be received.

queue: pointer to the queue control block to be used by Program Management to queue the signal so it will not be lost.

status: returned request status.

4.2.2.3 <u>PM#DESELECT_SIGNAL</u>

This request breaks the association of a signal with a queue. Further receptions of the specified signal will not result in those signals being items on the previously specified queue.

PM#DESELECT_SIGNAL (name, status)

name: the type and id of the signal as specified in a previous PM#SELECT_SIGNAL request.

status: returned request status.

4.2.2.4 <u>PM#STATUS_SIGNAL</u>

This request provides a way to determine if a particular signal has arrived. This is meaningful for the case of more than one signal associated with a queue. It can be determined if anything is on the queue by using the PM#STATUS_QUEUE request. It can be determined if a particular signal is on a queue by using the PM#STATUS_SIGNAL request.

PM#STATUS_SIGNAL (signal, queue, status)

name: type and id of a signal as previously specified in a PM#SELECT_SIGNAL request.

queue: returned pointer to queue control block, if any, as specified on a previous PM#SELECT_SIGNAL request.

status: returned request status.

4.2.2.5 <u>PM#DISABLE_SIGNALS</u>

This request is used to prevent loss of control due to interruption for signal processing for the requesting control point. This request does not prevent hardware interruptions.

PM#DISABLE_SIGNALS (status)

status: returned request status

4.2.2.6 PM#ENABLE_SIGNALS

This request is used to enable signal processing after a previous PM#DISABLE_SIGNALS request.

PM#ENABLE_SIGNALS (status)

status: returned request status.

4.2.2.7 PM#IDENTITY

This request is used to obtain the execution identity of the requestor. The execution identity may be the control point id, the task control block, the program control block, or the job control block.

PM#IDENTITY (to be supplied)

4.3 QUEUES

The queuing mechanism provided by Program Management is designed to allow the sending, storing, and retrieving of arbitrary data structures between asynchronous activities within a Job. The queuing facility will be used, for example, by the Signal mechanism to pass standard signals to the interested control points. An event may be associated with a queue so that an enqueue request on the queue would effect causation of the event. It is the responsibility of the owner of the QCB to put in the pointer to an associated ECB.

A queue is defined by a queue control block somewhere in the address space of the Job. It can be an LNS structure or somewhere in the job data base (stack, internal static, etc.) The format of a queue control block is shown below:

```
TYPE
    QUEUE_CONTROL_BLOCK = RECORD
        NUMBER_QUEUED: SEMAPHORE;
        ASSOCIATED_EVENT: ^EVENT_CONTROL_BLOCK,
        CHAIN_START: ^QUEUE_ITEM,
        CHAIN_END: ^QUEUE_ITEM,
        STORAGE_METHOD: (To Be Supplied)
    RECEND;
TYPE
    QUEUE_ITEM = RECORD
        QUEUE_THREAD: ^QUEUE_ITEM,
        DATA_LOCATION: ^SEQUENCE,
    RECEND.
```

The fields in the QUEUE_CONTROL_BLOCK and QUEUE_ITEM are described below:

NUMBER_QUEUED: This semaphore should have an initial value of zero. It indicates the number of items currently on the queue. Adding an item to the queue will do a SIGNAL_SEMAPHORE on this semaphore and getting an item from the queue will do a PM#WAIT_SEMAPHORE on this semaphore.

ASSOCIATED_EVENT: If the ^pointer is other than NIL it references an event control block which will be placed in the caused state whenever there are items on the queue and the cleared state whenever the queue is empty.

CHAIN_START: Pointer to the first item on the queue.

4.0 PROGRAM COMMUNICATIONS
4.3 QUEUES

CHAIN_END: Pointer to the last item on the queue.

STORAGE_METHOD: will indicate in some way where storage is to be acquired for new queue items.

QUEUE_THREAD: Thread of items on the queue.

DATA_LOCATION: Pointer to the data represented by the QUEUE_ITEM.

The actions performed by the queue request processors are described in the following decision table:

OPERATION	ENQUEUE				DEQUEUE			
NUMBER OF ITEMS ON QUEUE	0		>0		0		>0	
ASSOCIATED EVENT	Y	N	Y	N	Y	N	Y	N
Add item to queue	X	X	X	X				
Take item from queue							X	X
Suspend requesting process					X	X		
PM#CAUSE_EVENT	X							
PM#CLEAR_EVENT							*	

* if initial value is one.

4.0 PROGRAM COMMUNICATIONS
4.3 QUEUES

4.3.1 QUEUE REQUESTS

The queue requests provided by Program Management are as follows:

PM#ENQUEUE (queue, item, status)
PM#DEQUEUE (queue, item, status)
PM#STATUS_QUEUE (queue, status)

4.3.1.1 PM#ENQUEUE

The PM#ENQUEUE request adds a queue item to a queue and activates one process if there is one suspended on the queue.

PM#ENQUEUE (queue, item, status)

queue: pointer to the queue control block which defines the particular queue.

item: pointer to the queue item which is to be added to the queue.

status: returned request status.

4.3.1.2 PM#DEQUEUE

The PM#DEQUEUE request removes an item from a queue and returns the location of the item to the requestor. If the queue is empty at the time of the request, the requestor is suspended.

PM#DEQUEUE (queue, item, status)

queue: pointer to the queue control block which defines the particular queue.

item: returned pointer to item data. The queue item is no longer on the queue.

status: returned request status.

4.3.1.3 PM#STATUS QUEUE

The PM#STATUS_QUEUE request provides a way to determine if there are any items on the specified queue.

PM#STATUS_QUEUE (queue, status)

queue: pointer to the queue control block that defines the queue.

status: Indicates whether or not there are any items on the queue.

4.4 SEMAPHORES

Semaphores are system supported facilities which permit communication and synchronization among asynchronous activities within a job. A semaphore is represented by a semaphore control block somewhere in storage and two system requests to manipulate the control block. A semaphore is the most primitive facility supported by the operating system for synchronization and serialization of asynchronous activities. Semaphores are utilized by various system routines to serialize themselves and in the implementation of Locks and Queues.

A semaphore may be either an LNS variable or a structure in the Job data base (internal static, stack, etc.) The format is as shown below:

```
TYPE
    SEMAPHORE = RECORD
        VALUE: INTEGER,
        CHAIN: ^CONTROL_POINT,
    RECEND;


VAR
    ANY_SEMAPHORE: SEMAPHORE
```

The states of a semaphore are shown in the following table:

REQUESTED OPERATION	WAIT			SIGNAL		
INITIAL CONTENTS OF 'VALUE' (V)	<0	=0	>0	<0	=0	>0
Resultant contents of 'value' (V)	V-1	V-1	V-1	V+1	V+1	V+1
Add request process to chain and suspend	X	X				
Remove first process from thread and activate				X		
Process immediately continues			X	X	X	X

For a description of the properties of semaphores and some examples of their uses, see section titled Program Management Notes.

4.4.1 SEMAPHORE REQUESTS

The semaphore requests provided by Program Management are as follows:

```
PM#SIGNAL_SEMAPHORE (semaphore, status)
PM#WAIT_SEMAPHORE (semaphore, status)
```

4.4.1.1 PM#SIGNAL_SEMAPHORE

This request increments the 'value' of a semaphore by one. If the resultant value is less than or equal to zero, the process which has been waiting for the semaphore the longest is activated.

PM#SIGNAL_SEMAPHORE (semaphore, status)

semaphore: pointer to a structure of type semaphore

status: returned request status.

4.4.1.2 PM#WAIT_SEMAPHORE

This request decrements the 'value' of a semaphore by one. If the resultant value is less than zero, the requesting process is suspended.

PM#WAIT_SEMAPHORE (semaphore, status)

semaphore: pointer to a structure of type semaphore

status: returned request status.

4.4.2 INTRA-JOB LOCKS

Locks as such are not directly supported by the operating system as primitive requests since their function can be wholly replaced by semaphores. The simple two state lock is described in Denning's article in the section titled Program Management Notes.

A more flexible lock mechanism is proposed by the CODASYL Programming Language Committee Proposal ATG-71001.11. See the section titled Program Management Notes.

4.4.3 INTER-JOB SYNCHRONIZATION

The semaphore and lock mechanisms described above are for synchronization of asynchronous activities within a Job. There are two mechanisms which permit synchronization and communication <u>between</u> Jobs. One is the Signal facility described in 4.2. The other is the Compare and Swap hardware instruction which may be used on memory locations which are shared between Jobs. The Compare and Swap is externalized by two requests referencing a signature lock. The two coordinating jobs must be sharing a segment with an agreed upon word in that segment designated as the signature lock.

4.4.3.1 Signature Lock Requests

The signature lock requests provided by Program Management are as follows:

PM#SIGN_LOCK (lock, status)
PM#UNSIGN_LOCK (lock, status)

4.4.3.1.1 PM#SIGN_LOCK

This request is used to sign a signature lock with the Control Point id of the requestor. The request is rejected if the requesting control point already has anything locked via PM#SIGN_LOCK. Otherwise the request disables signal processing, does a compare swap on the signature lock word. If the compare swap is successful, returns leaving the Control Point id in the signature lock word. If not successful, enables signal processing and cycles.

PM#SIGN_LOCK (lock, status)

lock: pointer to the signature lock word in the shared segment.

status: returned request status.

4.4.3.1.2 PM#UNSIGN_LOCK

This request is used to unsign a signature lock by writing it with zeroes. Rejects if the requesting control point does not have it locked.

PM#UNSIGN_LOCK (lock, status)

lock: pointer to the signature lock word in the shared segment.

status: returned request status.

4.5 ON CONDITIONS

To be supplied.

5.0 PROGRAM MAINTENANCE

To Be Supplied

6.0 PROGRAM MANAGEMENT NOTES

6.1 COBOL LOCK NOTES

A flexible lock mechanism is proposed by the CODASYL Programming Language Committee Proposal ATG-71001.11. The lock defined there has four possible states and three operations defined on it as detailed in the following diagram:

REQUESTED OPERATION	UNLOCK				LOCK FOR SHARED USE				LOCK FOR EXCLUSIVE USE			
INITIAL STATE	U	L	M	E	U	L	M	E	U	L	M	E
Resultant state	U	U	M	U	L	M	M	E	E	L	M	E
Suspend requesting process								X		X	X	X
Activate suspended process if any		X		X								
Request error	X											
Process immediately continues	X	X	X	X	X	X	X		X			

The states correspond to:

U = unlocked.
L = locked for shared use by one process.
M = locked for shared use by multiple processes.

E = locked for exclusive use.

The following sample coding demonstrates how this fairly complex lock mechanism could be implemented by a run time support system or macros using the semaphore mechanism. However, this code example does not include the COBOL ATG 'AT LOCKED' immediate return option.

```
"Definition of a lock structure"

TYPE
    COBOL_LOCK = RECORD
        SHARED_COUNT : INTEGER
        EXCLUSIVE_LOCK : SEMAPHORE,
        SHARED_KEY : SEMAPHORE,
        EXCLUSIVE_KEY : SEMAPHORE,
    RECEND;

"Semaphore used to serialize lock/unlock procedures"

    LOCK_CONTROL : SEMAPHORE := [1, nIL];

"Unlock procedure used for both shared and"
"exclusive locks"

PROC COBOL_UNLOCK (REF I: COBOL_LOCK);
WAIT (LOCK_CONTROL);
IF I.EXCLUSIVE_LOCK.VALUE LT 1 THEN
    "If exclusively locked"
    SIGNAL (I.SHARED_KEY);
    SIGNAL (I.EXCLUSIVE_LOCK);
ELSE
    "LOCKED FOR SHARED USE"
    I.SHARED_COUNT := I.SHARED_COUNT-1;
    IF I.SHARED_COUNT EQ 0 THEN
        "activate waiters for exclusive lock"
        SIGNAL (I.EXCLUSIVE_KEY);
    IFEND;
IFEND;
SIGNAL (LOCK_CONTROL);
PROCEND COBOL_UNLOCK;



"Lock procedure for shared lock"

PROC COBOL_SHARED_LOCK (REF I: COBOL_LOCK);
LABEL START_LOOP;
```

6.0 PROGRAM MANAGEMENT NOTES
6.1 COBOL LOCK NOTES

```
START_LOOP: LOOP;
    WAIT (LOCK_CONTROL);
    IF I.EXCLUSIVE_LOCK.VALUE EQ 1 THEN
        "If not exclusively locked"
        IF I.SHARED_COUNT = 0 THEN
            "If unlocked prevent exclusive lock"
            WAIT (I.EXCLUSIVE_KEY);
        IFEND;
        I.SHARED_COUNT := I.SHARED_COUNT + 1;
        SIGNAL (LOCK_CONTROL);
        EXIT STARTLOOP;
    ELSE
        "If exclusively locked wait until unlocked"
        SIGNAL (LOCK_CONTROL);
        WAIT (I.SHARED_KEY);
        SIGNAL (I.SHARED_KEY);
    IFEND;
LOOPEND;
PROCEND COBOL_SHARED_LOCK;




"Lock procedure for exclusive lock"

PROC COBOL_EXCLUSIVE_LOCK (REF I: COBOL_LOCK);
LABEL START_LOOP;
START_LOOP:LOOP;
    WAIT (LOCK_CONTROL);
    IF I.EXCLUSIVE_LOCK.VALUE LT 1 THEN
        "If already exclusively locked"
        SIGNAL (LOCK_CONTROL);
        WAIT (I.EXCLUSIVE_LOCK);
        SIGNAL (I.EXCLUSIVE_LOCK);
    ELSE
        IF I.SHARED_COUNT >0 THEN
            "If locked for shared use"
            SIGNAL (LOCK_CONTROL);
            WAIT (I.EXCLUSIVE_KEY);
            SIGNAL (I.EXCLUSIVE_KEY);
        ELSE
            "If unlocked"
            WAIT (I.EXCLUSIVE_LOCK);
            "Prevent shared lock"
            PM#WAIT (I.SHARED_KEY);
            SIGNAL (LOCK_CONTROL);
            EXIT STARTLOOP;
```


6.0 PROGRAM MANAGEMENT NOTES
6.1 COBOL LOCK NOTES

```
            IFEND;
        IFEND;
    LOOPEND;
    PROCEND COBOL_EXCLUSIVE_LOCK.
```

6.2 SEMAPHORE NOTES

The following excerpt by Denning(1) very nicely describes the properties of semaphores and provides some examples of their uses:

"...A semaphore is an integer variable s with an initial value s0 $\geq$ 0 assigned on creation; associated with it is a queue Q, in which are placed the identifiers of processes waiting for the semaphore to be "unlocked." Two indivisible operations are defined on a semaphore s:

wait s:[s<- s - 1; if s < 0 the caller places himself in the queue Q, enters the waiting state, and releases the processor]

signal s:[s<- s + 1; if s $\leq$ 0 remove some process from Q, and add it to the work queue of the processors]

Semaphore values may not be inspected except as part of the wait and signal operation. If s < 0, then -s is the number of processes waiting in the queue Qs. Executing wait when s > 0 does not delay the caller, but executing wait when s $\leq$ 0 does, until another process executes a corresponding signal. Executing signal does not delay the caller. The programming for mutual exclusion using wait and signal is the same as for lock and unlock, with x0 = 1 (wait replaces lock, and signal replaces unlock)...

"Synchronization

In a computation performed by cooperating processes, certain processes may not continue their progress until information has been supplied by others. In other words, although program-executions proceed asynchronously, there may be a requirement that certain program-executions be ordered in time.

---

1 Third Generation Computer Systems', Peter J. Denning, Computer Surveys, Vol. 3, No. 4, December, 1971, pp. 199-201.

This is called synchronization. The precedence constraints existing among processes in a system express the requirement for synchronization. Mutual exclusion is a form of synchronization in the sense that one process may be blocked until a signal is received from another. The wait and signal operations, which can be used to express all forms of synchronizations, are often called synchronizing primitives.

"An interesting and important application of synchronization arises in conjunction with cooperating cyclic processes. An example made famous by Dijkstra is the "producer/consumer" problem, an abstraction of the input/output problem. Two cyclic processes, the producer and the consumer, share a buffer of $n > 0$ cells, the producer places items there for later use by the consumer. The producer might, for example, be a process that generates output one line at a time, and the consumer a process that operates the line printer. The producer must be blocked from attempting to deposit an item into a full buffer, while the consumer must be blocked from attempting to remove an item from an empty buffer. Ignoring the details of producing, depositing, removing, and consuming items, and concentrating solely on synchronizing the two processes with respect to the conditions "buffer full" and "buffer empty", we arrive at the following abstract description of what is required. Let a1a2...ak be a system action sequence for the system consisting of the producer and consumer processes. Let p(k) denote the number of times the producer has deposited an item among the actions a1a2...ak, and let c(k) denote the number of times the consumer has removed an item from among the action a1a2...ak. It is required that

$$0 \le p(k) - c(k) \le n \quad (1)$$

for all k. The programming that implements the required synchronization (Eq. 1) is given below; x and y are semaphores with initial values x0 = 0 and y0 = n:

```
   pro:produce item;     con:wait x;
       wait y;               remove item;
       deposit item;         signal y;
       signal x;             consume item;
       goto pro;             goto con;
```

To prove that Eq. 1 holds for these processes, suppose otherwise. The either c(k) > p(k) or p(k) > c(k) +n. However, c(k) > p(k) is impossible since it implies that the number of completed wait x exceeds the number of completed signal x, thus contradicting x0 = 0. Similarly, p(k) > c(k) + n is also impossible since it implies that the number of completed wait y

exceeds by more than n the number of completed signal y, thus contradicting y0 = n.

"Another application of synchronization is the familiar "read-acknowledge" form of signaling, as used in sending a message and waiting for a replay (B5). Define the semaphores r and a with initial values r0 = a0 = 0; the programming is of the form:

```
       IN THE SENDER          IN THE RECEIVER
             .                      .
             .                      .
             .                      .
    generate message;         wait r;
    signal r;                 obtain message;
    wait a;                   generate reply;
    obtain reply;             signal a;
             .                      .
             .                      .
             .                      .
```

"...As a final example, let us consider how the synchronizing primitives can be used to describe the operation of an interrupt system. Typically, the interrupt hardware contains a set of pairs of flipflops, each pair consisting of a "mask flipflop" and an "interrupt flipflop." the states of the flipflops in the ith pair are denoted by mi and xi, respectively. The ith interrupt is said to be "disabled" (masked off) if mi = 0, and "enabled" if mi = 1. When the hardware senses the occurrence of the ith exceptional condition Ci, it attempts to set xi = 1; if mi = 0, the setting of xi is delayed until mi = 1. The setting of xi is supposed to awaken the ith "interrupt-handler process" Hi, in order to act on the condition Ci. By regarding mi and xi as hardware semaphores with initial values mi = 1 and xi = 0, we can describe the foregoing activities as an interprocess singaling problem:

```
       IN HARDWARE
          Ci occurs:wait mi;
                    signal xi;
                    signal mi;
            disable:wait mi;
             enable:signal mi;

       IN INTERRUPT HANDLER Hi
             start:wait xi;
                   process interrupt;
                   goto start;
```