

AD-A094 569

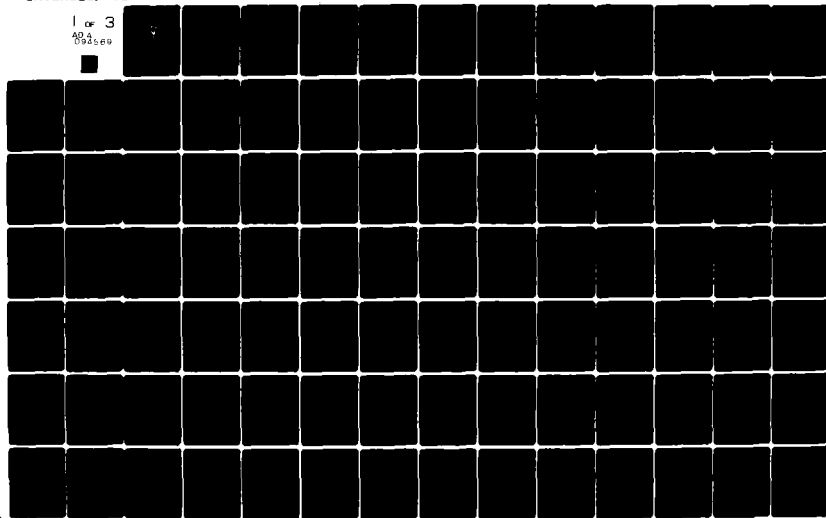
NAVAL POSTGRADUATE SCHOOL MONTEREY CA
IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A SECURE ARCHIVAL STOR--ETC(U)
SEP 80 J T WELLS

F/6 9/2

UNCLASSIFIED

NL

1 of 3
AD-A
098569



② LEVEL II

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD A094569



DTIC
ELECTE
FEB 4 1981
S B

THESIS

IMPLEMENTATION OF SEGMENT MANAGEMENT
FOR A SECURE ARCHIVAL STORAGE SYSTEM

by

John Timothy Wells

September 1980

Thesis Advisor:

R. R. Schell

DDC FILE COPY

proved for public release; distribution unlimited

81 2 04 003

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-4094569	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Implementation of Segment Management for a Secure Archival Storage System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; September 1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John Timothy Wells		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE September 1980
		13. NUMBER OF PAGES 242
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) security kernel, operating systems, segmentation, information security, archival storage, security policy		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This thesis presents an implementation of segment management for the security kernel of a secure archival storage system. The basis for this implementation is a family of secure, distributed, multi-microprocessor operating systems designed to provide multi- level internal computer security and controlled sharing of data among authorized users. This implementation provides address space management for individual processes, based on segmentation as a		

memory management scheme. Non-discretionary information security is provided through enforcement of a security policy based on a lattice structure that allows flexibility in representing different security policies; the Department of Defense (DoD) security classification system is the security policy represented in this thesis. Implementation was completed on the ZILOG Z8000 micro-processor.

Approved by _____ Date _____	
Dist _____	Special _____

Approved for public release; distribution unlimited.

Implementation of Segment Management
for a
Secure Archival Storage System

by

John T. Wells
Lieutenant Commander, United States Navy
B.S., University of Mississippi, 1970

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1980

Author

John T. Wells

Approved by:

Roger R. Schell

Thesis Advisor

John A. Cochran

Second Reader

W. M. Woods

Chairman, Department of Computer Science

W. M. Woods

Dean of Information and Policy Sciences

ABSTRACT

This thesis presents an implementation of segment management for the security kernel of a secure archival storage system. The basis for this implementation is a family of secure, distributed, multi-microprocessor operating systems designed to provide multilevel internal computer security and controlled sharing of data among authorized users. This implementation provides address space management for individual processes, based on segmentation as a memory management scheme. Non-discretionary information security is provided through enforcement of a security policy based on a lattice structure that allows flexibility in representing different security policies; the Department of Defense (DoD) security classification system is the security policy represented in this thesis. Implementation was completed on the ZILOG Z8000 microprocessor.

TABLE OF CONTENTS

I.	INTRODUCTION.....	10
A.	BACKGROUND.....	11
B.	SECURE ARCHIVAL STORAGE SYSTEM OVERVIEW.....	14
1.	Levels of Abstraction.....	14
2.	Level Three - Host Computer(s).....	15
3.	Level Two - Supervisor.....	17
4.	Gate Keeper Module.....	19
5.	Level One - Kernel.....	21
a.	Segment Manager.....	21
b.	Non-Discretionary Security Module.....	22
c.	Event Manager.....	23
d.	Traffic Controller.....	24
e.	Inner Traffic Controller.....	25
f.	Memory Manager.....	28
6.	Level Zero - Hardware.....	31
C.	STRUCTURE OF THE THESIS.....	32
II.	SEGMENT MANAGEMENT FUNCTIONS.....	36
A.	BASIC CONCEPTS/DISCUSSION.....	36
1.	Segmentation.....	36
2.	Data Sharing.....	37
3.	Information Security.....	39
a.	Basic Security Principles.....	40
b.	Lattice Model Abstraction.....	43
c.	Examples.....	44

d. Applications to the SASS.....	47
B. SEGMENT MANAGER.....	48
1. Function.....	48
2. Database.....	50
C. NON-DISCRETIONARY SECURITY MODULE.....	54
D. MEMORY MANAGER.....	55
1. Function.....	55
2. Databases.....	56
E. SUMMARY.....	58
III. SEGMENT MANAGEMENT IMPLEMENTATION.....	60
A. IMPLEMENTATION ISSUES.....	60
1. Interprocess Messages.....	61
2. Structures as Arguments.....	63
3. Reentrant Code.....	63
4. Process Structure of Memory Manager.....	64
5. Per-Process Known Segment Table.....	64
6. DER Handle.....	65
B. SEGMENT MANAGER MODULE.....	65
1. Create a Segment.....	66
2. Delete a Segment.....	69
3. Make a Segment Known.....	70
4. Make a Segment Unknown (Terminate).....	73
5. Swap a Segment In.....	75
6. Swap a Segment Out.....	75
C. NON-DISCRETIONARY SECURITY MODULE.....	76
1. Equal Classification Check.....	78

2. Greater or Equal Classification Check.....	78
D. DISTRIBUTED MEMORY MANAGER MODULE.....	80
1. Description of Procedures.....	80
2. Interprocess Communication.....	83
E. SUMMARY.....	85
IV. CONCLUSIONS AND FOLLOW ON WORK.....	98
APPENDIX A--SEGMENT MANAGER PLZ/SYS LISTINGS.....	100
APPENDIX B--SEGMENT MANAGER PLZ/ASM LISTINGS.....	110
APPENDIX C--DIST. MEMORY MANAGER PLZ/SYS LISTINGS.....	131
APPENDIX D--DIST. MEMORY MANAGER PLZ/ASM LISTINGS.....	141
APPENDIX E--NON-DISC. SECURITY PLZ/SYS LISTINGS.....	159
APPENDIX F--NON-DISC. SECURITY PLZ/ASM LISTINGS.....	161
APPENDIX G--SUMMARY OF REFINEMENTS.....	163
APPENDIX H--SEGMENT MANAGEMENT DEMONSTRATION.....	165
APPENDIX I--DEMONSTRATION LISTINGS.....	169
LIST OF REFERENCES.....	239
INITIAL DISTRIBUTION LIST.....	241

LIST OF FIGURES

1. SASS System Overview.....	16
2. Active Process Table.....	26
3. Virtual Processor Table.....	29
4. Summary of Extended Instruction Sets.....	34
5. Summary of Kernel Databases.....	35
6. Known Segment Table.....	53
7. Memory Management Unit Image.....	59
8. Memory Manager - CPU Table.....	59
9. Initialized Active Process Table.....	86
10. Initialized Virtual Processor Table.....	87
11. Initialized Known Segment Tables.....	88
12. Initialized Memory Management Unit Image.....	89
13. Initialized Process Stack Segments.....	90
14. Linker and Imager Command Lines.....	91
15. Load Command Lines and Register Initialization.....	92
16. Generated Output.....	93

ACKNOWLEDGEMENTS

This research is sponsored in part by the Office of Naval Research Project NR 337-005, monitored by Mr. Joel Trimble.

I wish to express my deepest appreciation to my advisor Lt. Col Roger Schell. His patience and assistance were invaluable. Thanks also to my reader Professor Lyle Cox and to Lcdr. Ed Moore, Lcdr. Steve Reitz, and Lt. Al Gary. Finally, special thanks to my closest friend (and wife) Susan, who always supports me, whatever the endeavor.

I. INTRODUCTION

This thesis addresses the implementation of the segment management functions of an operating system known as the Secure Archival Storage System or SASS. This system, with full implementation, will provide: (1) multilevel secure access to information (files) stored in a "data warehouse" for a network of multiple host computers, and (2) controlled data sharing among authorized users. The correct performance of both of these features is directly dependent upon the proper implementation of the segment management functions addressed in this thesis. The issue of access to sensitive information is addressed by the Non-Discretionary Security Module, which mediates all non-discretionary access to information. Sharing of information is accomplished chiefly through the properties of segmentation, the SASS memory management scheme that is supported by the Memory Manager Module and the Segment Manager Module. The implementation of segment management for SASS is thus integral to the attainment of the two key goals that SASS was designed to achieve. This implementation addresses the Non-Discretionary Security, Distributed Memory Manager (the interface to the Memory Manager Process), and Segment Manager modules.

A. BACKGROUND

O'Connell and Richardson provided the design for a family of secure, distributed, multi-microprocessor operating systems from which the subset, SASS, was later derived [6]. In their work, two of the primary motivations were to provide a system that (1) effectively coordinated the processing power of microprocessors and (2) provided information security.

The basis for emphasis on utilization of microprocessors is not purely that of replacing software with more powerful (and faster) hardware (microprocessors) but is also an economic issue. Software development and computing operations are becoming more and more expensive, putting further pressure on system designers to increasingly utilize people solely for system functions that computers cannot perform in a cost effective manner. Microcomputers, on the other hand, are becoming less and less expensive and are, therefore, increasingly being used for more functions.

The need for information security has been gradually recognized as the uses of computers have expanded. As security needs for specific computer systems have been recognized, attempts have been made to modify the existing systems to provide the desired security. The results have been systems that could not be certified as secure and/or which have failed to resist penetration efforts, i.e. systems which, in effect, did not provide adequate

information security. It has become clear that, in order to be certifiably secure, a computer system must have security designed in from first principles [10,11]. Such is the case with SASS. Information security was and continues to be a chief design feature. Integral to the design goal of information security were two related goals. One of these goals was to provide multilevel controlled access to a consolidated "warehouse" of data for a network of multiple host computers. The other key goal was to provide for controlled sharing among the computer hosts.

A brief background of prior work relative to SASS follows. O'Connell and Richardson originated the design of a secure family of operating systems. Their design provided two basic parts for their system -- the supervisor (to provide operating system services) and the kernel (to provide for physical resource management). The design of the SASS supervisor was completed by Parks [7]. No implementation or further design effort on the supervisor has followed, to date. The initial design of the kernel was completed by Coleman [1]. That design described the kernel in terms of seven modules:

1. Gate Keeper Module -- provided for ring-crossing mechanism and thus isolation of the kernel.
2. Segment Manager Module -- provided for management of segmented virtual memory.
3. Traffic Controller Module -- multiplexed processes onto virtual processors and supports the inter-

process communication primitives Block and Wakeup.
Block and Wakeup.

4. Non-Discretionary Security Module -- mediated non-discretionary security access attempts.
5. Inner Traffic Controller Module -- multiplexed virtual processors onto real processors and provided the Kernel synchronization primitives Signal and Wait.
6. Memory Manager Module -- managed main memory and secondary storage.
7. Input-Output Manager -- managed the moving of information to external devices outside the boundaries of the SASS.

Refinement of the kernel design and partial implementation was completed by Gary and Moore [4] in conjunction with Reitz [9]. The resultant description of the kernel as a result of their work was:

1. Gate Keeper Module
2. Segment Manager Module
3. Event Manager Module -- worked with the Traffic Controller to manage the "event data" associated with the IPC mechanism of eventcounts and sequencers.
4. Non-Discretionary Security Module
5. Traffic Controller Module -- replaced Block and Wakeup with Advance and Await (to implement Supervisor IPC mechanism of eventcounts and sequencers).
6. Memory Manager Module
7. Inner Traffic Controller Module

Reitz implemented the Traffic Controller Module and Inner Traffic Controller Module. Gary and Moore completed a

detailed design of the Memory Manager, originated the Memory Manager code (written predominantly in PLZ/SYS), selected a thread of the code, hand compiled it into PLZ/ASM and ran it on the Z8000 developmental module.

The design and implementation works mentioned above provided the design base for this implementation. Refinements were made as needed and are discussed in Appendix G of this thesis. A broader description of the current state of SASS will be provided in the next section.

B. SECURE ARCHIVAL STORAGE SYSTEM OVERVIEW

This section presents a brief summary of the current design state of the Secure Archival Storage System. The purpose of this summary is to provide continuity (interface) between this and previous work relative to SASS, and to enhance understanding of the evolution of the more detailed and system specific information provided later in this thesis.

1. Levels of Abstraction

The original design for a family of secure, distributed operating systems (which was the basis for the development of SASS) used effectively the concept of levels of abstraction as a design methodology tool. Just as this tool allows for clarity and simplification in conceptualizing and designing a system, it also enhances the ability to clearly and succinctly describe that system's

design. Thus, an abstract system overview (description) of SASS will be presented here. Figure 1 represents that overview (illustrated for a single host system for clarity). There are four levels of abstraction:

Level 3 -- the Host computer systems

Level 2 -- the Supervisor

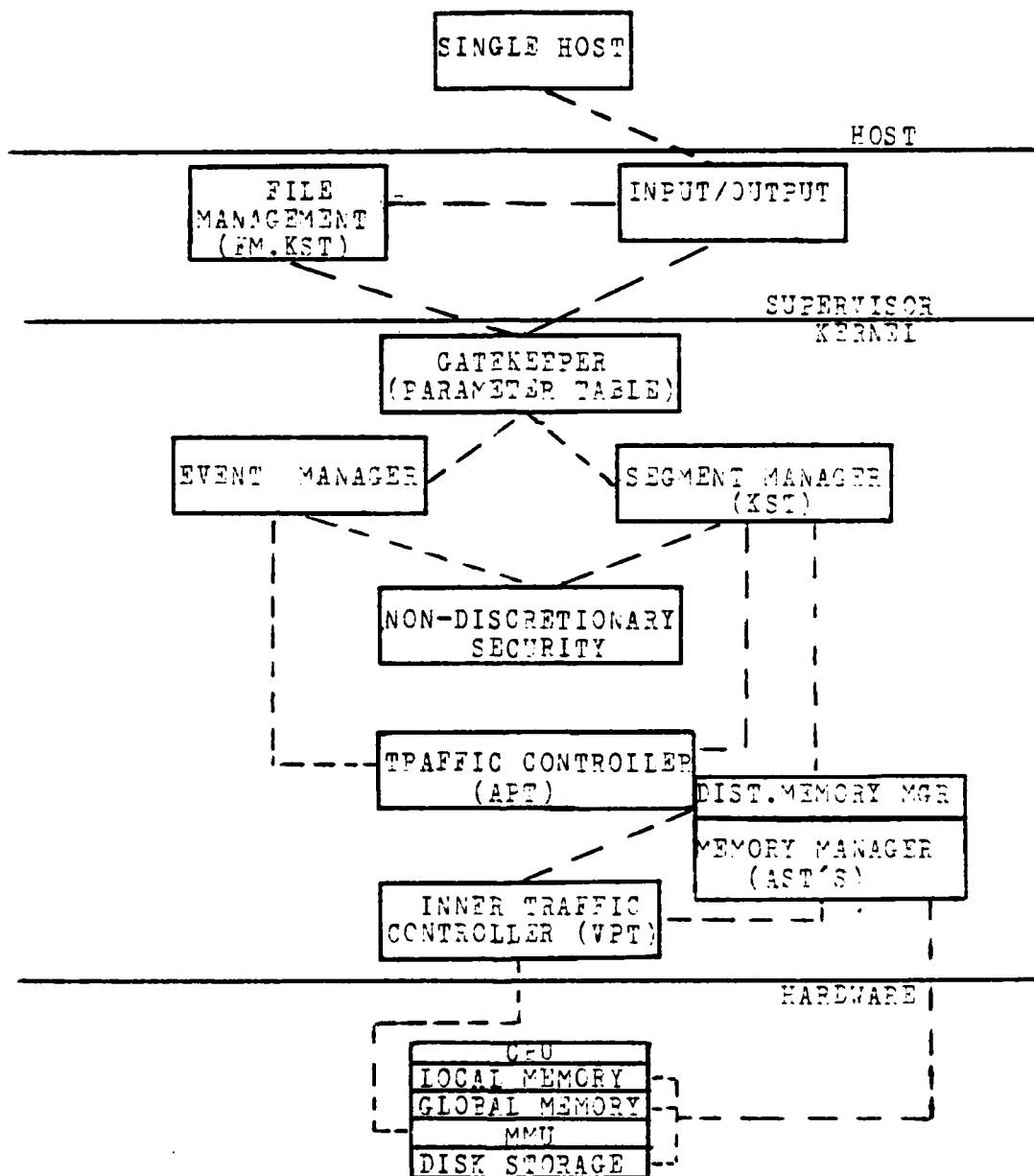
Level 1 -- the Kernel

Level 0 -- the Hardware

The Gate Keeper Module is logically the boundary between the Supervisor and the Kernel and thus will not be discussed within either of these levels, but rather separately.

2. Level Three - Host Computer(s)

Level three consists of the Host computer systems. There may be a variable number of host computers of any type (e.g., micro, mini, etc). Each host may be used to create and manipulate files of a fixed, predetermined degree of sensitivity (or security classification). Once a user of a host computer system completes work on a particular file, they can permanently store that file on the SASS (and, of course, may later again access the same file by requesting the SASS to provide it to them). Each host computer system is individually wired to an I/O port of the Z8001. Each of the ports has fixed access level.



Note: Databases are shown in parenthesis

Figure 1. SASS System Overview (Single Host)

If a multilevel secure Host desires to handle data at two levels (e.g., secret and unclassified), it will use two connections to the SASS. Physical and/or cryptographic protection of the hardwire connections is assumed.

3. Level Two - Supervisor

Level two is the Supervisor. A proper description of the Supervisor is "that component of the SASS that executes in the outer, less privileged domain (normal mode) of the 28001 microprocessor, and is responsible for the SASS - Host computer interface". Integral to this description (and the Kernel's description) is the concept of a "domain", that can be described using four other terms that also need to be defined: "process", "address space", "segment", and "segmentation". Madnick and Donovan [5] define a process as the locus of points of a processor executing a collection of programs; the collection of programs and data that are accessed in a process forms that process's address space. A segment is defined as a logical grouping of information, such as a subroutine, array, or data area, and segmentation is defined as the technique for managing segments of an address space. It is convenient then, since the SASS uses segmentation as a memory management scheme, to more specifically define a SASS process address space as the collection of segments that are accessed (or are accessible) in that process. A domain is conceptualized in SASS due to the necessity to isolate the Kernel from all possible

outside influences. To achieve this, a process' address space is divided into a hierarchical arrangement of segment accessibility, viz., a set of hierarchical protection domains called protection rings. In SASS, there are two domains implemented (and necessary as a minimum): the Supervisor domain and the Kernel domain. The Z8001 microprocessor provides the SASS with two execution modes that, along with Kernel software, implement these domains: a system (Kernel) mode that provides access to all segments (and machine instructions), and a normal (Supervisor) mode that provides access to a subset of the segments (and machine instructions). Thus, the Supervisor operates in the outer or less privileged domain.

The Supervisor contains those segments of the system that are necessary to perform the SASS - Host computer system interface (construct and manage a file hierarchy and control I/O between the SASS - Host). It is built upon the Kernel and performs the Host's requests by calls to the Kernel (the calls are validated by the Gate Keeper prior to invocation of Kernel functions). Two surrogate processes, input/output (I/O) and file management (FM), are assigned to each Host computer system at system generation. The FM process directs all interaction between the SASS and a Host computer system. Specific functions include the management of the Host's file hierarchy (using the FM Known Segment Table (FM_KST) as a database) and discretionary security

access management (checking and maintaining an Access Control List (ACL) for each file within the file hierarchy). Controlling discretionary security with an ACL allows authorized users to specify who may use segments (files) within the confines of the non-discretionary security policy. Discretionary security will be defined and discussed in more detail in chapter II.

The I/O process acts in a slave mode to the FM process and is responsible for all input/output between the Supervisor and the host computer systems. Data is transferred between the Host and the SASS via fixed size "packets" (a grouping of data in a specified format). To transmit and receive packets between the Host and the SASS a "protocol" (or formal passing method) must exist between them. The I/O process is responsible for the SASS-Host protocol (Parks [7] designed a multi-packet protocol). Parks provides a detailed description of the Supervisor as it was originally designed.

4. Gate Keeper Module

The Gate Keeper is a software ring-crossing mechanism that provides for the isolation of the Kernel (viz., making the Kernel procedures tamperproof). The notion of a "ring-crossing" mechanism is an extension of the previous discussion of domains since "protection rings" is simply another term for hierarchial domains (such as the SASS arrangement of the Kernel and the Supervisor). All

calls to the distributed kernel and IPC with the Memory Manager must pass through the Gate Keeper (viz., it is the sole entry point into the Kernel from the Supervisor). The Gate Keeper is a trap handler; when invoked by the supervisor domain of a process, it must save the supervisor domain registers and stack pointer. The argument list provided by the supervisor domain's call (included in this list must be the identity of the kernel domain function (procedure) being called) is validated and, if correct, results in invocation of the appropriate procedure. Hardware preempt interrupts are masked upon entry into the Kernel. When returning (exiting the Kernel) the following actions occur: (1) software virtual preempt interrupts are unmasked (if a virtual preempt interrupt has occurred, the Traffic Controller's virtual interrupt handler is called vice the Kernel being exited), (2) hardware interrupts are unmasked, (3) the return arguments are passed to the Supervisor, and (4) the Supervisor domain stack pointer and registers are restored, returning the execution point to the Supervisor domain. An error code is returned and the Kernel is not invoked when an invalid call is encountered by the Gate Keeper. The database of the Gate Keeper is the Parameter Table. This table contains an entry for each permitted kernel function (e.g., Create_Segment, Delete_Segment, etc.) and is used to validate the correctness of the range (size) of the parameters passed.

5. Level One - Kernel

Level one is the Security Kernel (or Kernel). The Security Kernel in the inner or most privileged domain (system mode of the Z8001) and is responsible for managing the real resources of the hardware system (viz., memory, microprocessor, external devices, and input/output ports), and for enforcing the non-discretionary security policy for the SASS. The Kernel is divided into two major components. The first is the distributed kernel, i.e., the modules in the Kernel whose segments are placed in (or distributed in) the address spaces of each Supervisor process; the distributed kernel consists of the Gate Keeper (already discussed), the Segment Manager, the Event Manager, the Traffic Controller, and the Inner Traffic Controller. The second component is the non-distributed kernel and consists of the asynchronous memory manager process (which is contained entirely within the Kernel address space). There is a memory manager process for each hardware processor in the SASS. The following section will identify and briefly describe each of the Kernel's distributed and non-distributed system components.

a. Segment Manager

The Segment Manager (the focal point of this thesis) is a component of the distributed kernel; its function is the creation and management of a segmented virtual memory for the process. Actual memory management

functions are completed via calls for IPC to the Memory Manager process. Calls to (viz., entries into) the Segment Manager are received via the Gate Keeper from the Supervisor. These entries (viz., extended instructions) are:

1. Create_Segment -- add a new segment to the SASS.
2. Delete_Segment -- remove a segment from the SASS.
3. Make_Known -- add a segment to a process' address space.
4. Terminate -- remove a segment from a process' address space.
5. SM_Swap_In -- move a segment from secondary storage to main memory.
6. SM_Swap_Out -- move a segment from main memory to secondary storage.

The process local database used by the Segment Manager is the Known Segment Table (KST). The KST contains entries for all segments in the address space of that process. The Segment Manager will be described in more detail in chapters II and III.

b. Non-Discretionary Security Module

The Non-Discretionary Security (NDS) Module is a component of the distributed kernel; its function is the enforcement of the non-discretionary security policy in effect in the SASS. Although the implementation presented in this thesis reflects the DoD non-discretionary security policy, any security policy that can be represented by a

lattice structure may be similarly implemented. To implement a different policy (e.g., Privacy Act or a local policy) requires only replacement of the Non-Discretionary Security Module (viz., the modules calling it can be left intact with no changes required to them). The NDS Module creates the extended instruction set CLASS_EQ and CLASS_GE. The Non-Discretionary Security Module and the information security concepts which form its basis will be discussed in more detail in chapters II and III.

c. Event Manager

The Event Manager is a component of the distributed kernel; it is invoked by the Supervisor processes via the Gate Keeper. This module's function is to manage event data. Event data is associated with a global object (called an eventcount). An eventcount is a count of the number of events (e.g., the number of read or write accesses of a segment) that have occurred so far in the execution of a system. In SASS, as a naming convention, each Supervisor segment has two eventcounts associated with it. These eventcounts (Instance1 and Instance2) are stored in a Memory Manager database. The Event Manager creates the extended instruction set READ and TICKET; they are based on the mechanism of eventcounts and sequencers (used for the synchronization of concurrent processes). READ is a call that returns the current value of the eventcount. TICKET, using a nondecreasing integer called a sequencer (also

associated with each Supervisor segment), provides a complete time ordering of possibly concurrent events. Each invocation of the function TICKET increments the value of the sequencer and returns it to the caller. The eventcounts/sequencer synchronization mechanism is described in detail by Reed and Kanodia [8] while an excellent abridged discussion is presented by Gary and Moore [4].

d. Traffic Controller

The Traffic Controller (TC) is a component of the distributed kernel; it is responsible for multiplexing processes onto virtual processors. A virtual processor is a data structure that contains a complete description of a process in execution on a physical processor at a given instant. A "complete description" is defined to consist of the execution point or current CPU state and the address space (set of segments accessible by that process) of the process in execution. The Traffic Controller also creates the extended instructions ADVANCE and AWAIT which are used to implement eventcounts and sequencers, the inter-process communication (IPC) mechanism invoked by the Supervisor, and the extended instruction PROCESS_CLASS. PROCESS_CLASS is invoked by the Segment Manager and returns the label (classification) of the current process. The Traffic Controller is half of a two level traffic controller; the other half is the Inner Traffic Controller, which multiplexes the virtual processors onto physical processors.

The database for the Traffic Controller is the Active Process Table (APT), a fixed size, system wide database, that contains a permanent entry for each Supervisor process created at system generation (in the SASS the processes are then active for the life of the system). The APT structure is shown in figure 2. The scheduling algorithms and a detailed discussion of the current state of the Traffic Controller are provided by Reitz [9].

e. Inner Traffic Controller

The Inner Traffic Controller (ITC) is a component of the distributed kernel; it is the other half of the two level traffic controller and its function is to multiplex (temporarily bind) virtual processors (VP) to the real processors of the system; a design choice was made to provide each system CPU with a small fixed set of virtual processors. Two of the VP's are the Memory Manager VP and the Idle VP (the latter is permanently bound to the lowest priority virtual processor and is scheduled by the ITC only when there is no useful work for the CPU). The remaining VP's have Supervisor processes temporarily bound on them by the Traffic Controller. Another function of the ITC is to furnish inter-process services for VP's in the kernel ring. This is done by providing the primitives SIGNAL and WAIT, that are used by processes in the Kernel ring to communicate with other Kernel ring processes.

LOCK	
RUNNING LIST	PROCESS_ID
VP_ID →	:
READY LIST	
BLOCKED LIST	

AP_INDEX

DER	ACCESS_CLASS	STATE	NEXT_AP	EVENTCOUNT HANDLE INSTANCE COUNT

Figure 2. Active Process Table

```

APT          RECORD [LOCK          WORD
                    RUNNING_LIST  RUNNING_ARRAY
                    READY_LIST    WORD
                    BLOCKED_LIST  WORD
                    AP            ARRAY
                                [NR_PROCESSES AP_TABLE]
                    ]

AP_TABLE     RECORD [NEXT_AP      AP_POINTER
                    PER           ADDRESS
                    ACCESS_CLASS  LONG
                    STATE         INTEGER
                    NEXT_AP       WORD
                    EVENT_COUNT   EVENT_TABLE
                    ]

EVENT_TABLE  RECORD [HANDLE       WORD
                    INSTANCE      WORD
                    COUNT         WORD
                    ]

AP_POINTER   WORD

ADDRESS      WORD

```

Figure 2. Active Process Table (continued)

This is the mechanism used within the Kernel to provide multiprogramming (process switching). The ITC Module creates the extended instruction set: SIGNAL, WAIT, SWAP_VDBR, IDLE, SET_PREEMPT, TEST_PREEMPT, and RUNNING_VP. The functions of SIGNAL and WAIT have been discussed already. SWAP_VDBR provides the TC with a means to schedule processes on the currently running VP. IDLE loads an "idle" process on the currently running VP. SET_PREEMPT sets the virtual preempt interrupt flag on a specified VP (specified by the TC). TEST_PREEMPT provides the virtual preempt unmasking mechanism that is executed each time a process tries to move from the Kernel to the Supervisor domain. The database used by the Inner Traffic Controller is the Virtual Processor Table (VPT). There is one system wide table with entries for each physical processor in the system. The VPT for a single processor system (such as SASS) is shown in figure 4. The scheduling algorithms and a detailed discussion of the current state of the Inner Traffic Controller are provided by Reitz[9].

f. Memory Manager

The Memory Manager Module is the only component in the non-distributed kernel. There is a Memory Manager process dedicated to each physical processor (CPU) in the system.

LOCK	
RUNNING LIST	
READY LIST	
FREE_LIST	

VP_INDEX
↓

DBR_NO	PRI	STATE	IDLE FLAG	PRFEMPT	PHYS PROCESSOR	NEXT_ VP	MSG LIST

MSG_INDEX
↓

MESSAGE	SENDER	NEXT_MSG

Figure 3. Virtual Processor Table

VPT	RECORD	[LOCK RUNNING_LIST READY_LIST FREE_LIST VP MSG_0	WORD VP_INDEX VP_INDEX MSG_INDEX ARRAY [NR_VP VP_TABLE] ARRAY [NR_VP MSG_TABLE]
VP_TABLE	RECORD	[DEF PRI STATE IDLE_FLAG PREEMPT PHYS_PROCESSOR NEXT_READY MSG_LIST]	ADDRESS WORD WORD WORD WORD WORD VP_INDEX MSG_INDEX
MESSAGE	ARRAY	[16	BYTE]
ADDRESS	WORD		
VP_INDEX	INTEGER		
MSG_INDEX	INTEGER		

Figure 3. Virtual Processor Table (continued)

The Memory Manager is responsible for managing the real memory resources of the system, viz., local and global main memory and secondary storage. The memory manager manages the local and global memory in such a way as to control bus contention in the multi-microprocessor environment. Thus, each CPU has its own local memory to store process local segments and there is a global memory to which every CPU has access and in which shared, writeable segments must be stored. This requirement is to ensure that a current copy is always accessed for a shared, writeable segment. To keep bus contention between processors that access global memory to a minimum, whenever possible (viz., in all cases but shared, writeable segments) segments are to be stored in local memory. The Memory Manager has several databases, primary of which are the system wide Global Active Segment Table (G_AST) and the per processor Local Active Segment Table (L_AST). A more detailed description of the Memory Manager Module is presented in chapters II and III.

6. Level Zero - Hardware

The Z8001 microprocessor, Z8010 Memory Management Unit (MMU), local and global memories, and secondary storage form the SASS' basic hardware group. Since the design calls for SASS to exist in a multi-microprocessor environment, there will be multiple copies of some elements of the group, e.g., CPU, local memory. The Z8001 microprocessor is a

register oriented machine that has sixteen 16-bit general purpose registers. When operated with the MMU, the desired capabilities of memory segmentation, multiple domains, and process switching are realized. The MMU consists of a set of registers (64) to implement the descriptor list (or descriptor segment); viz., each register contains the descriptor (containing the attributes) of a particular segment. Zilog [14] provides a detailed description of the Z8001 microprocessor and Zilog [15] describes the Z8010 MMU.

C. STRUCTURE OF THE THESIS

This thesis describes the implementation of the segment management functions for the SASS. The design "base" evolved from the original secure family of operating systems identified and designed by O'Connell and Richardson. A block structured language, PLZ/SYS, was used in this and previous design efforts, while implementation was completed using PLZ/ASM assembly code. PLZ/SYS is described by Snook [12] and Conway [2] while PLZ/ASM is described by Zilog [13]. A compiler for PLZ/SYS to PLZ/ASM code translation was not available. As a result, implementation included the added step of manual translation of PLZ/SYS code to PLZ/ASM code to facilitate testing and debugging.

In this chapter an introduction to SASS was provided through discussion of its background and an overview of the entire system. A summary of the extended instruction sets

created by the Kernel components and a summary of the Kernel databases is presented in figures 4 and 5.

Chapter II of this thesis will present a description of the segment management functions in SASS. Discussion of the theory behind information security and its implications to SASS is also provided. The modules encompassed by segment management will be discussed in terms of their design, functional purpose, and database descriptions.

Chapter III presents the implementation of segment management (viz., the segment manager, non-discretionary security, and distributed memory manager modules). Description of design and implementation criteria, and choices made during implementation are discussed in this chapter.

Chapter IV provides the conclusions reached, status of research, and recommendations relative to continuation and extension of the work.

Appendices include PLZ/ASM code for the modules, the program listings for the Segment Manager demonstration and a summary of the refinements made to previous design/code relative to SASS.

MODULE	INSTRUCTION SET	
Segment Manager	Create_Segment	Delete_Segment
	Make_Known	Terminate
	SM_Swap_In	SM_Swap_Out
Event Manager	Read	Ticket
Non-Discretionary Security Traffic Controller	Class_EQ	Class_GE
	Advance	Await
	Process_Class	
Inner Traffic Controller	Signal	Wait
	Swap_VDER	Idle
	Set_Preempt	Test_Preempt
	Running_VP	
Memory Manager	MM_Create_Entry	MM_Delete_Entry
	MM_Activate	MM_Deactivate
	MM_Swap_In	MM_Swap_Out

Figure 4. Extended Instruction Sets

MODULE	DATABASE
Gate Keeper	Parameter Table
Segment Manager	Known_Segment_Table (KST)
Traffic Controller	Active_Process_Table (APT)
Inner Traffic Controller	Virtual_Processor_Table (VPT)
Memory Manager	Global_Active_Segment_Table (G_AST)
	Local_Active_Segment_Table (L_AST)
	Memory_Management_Unit_Image (MMU_Image)
	Alias_Table
	Disk_Bit_Map
	Global_Memory_Bit_Map
	Local_Memory_Bit_Map

Figure 5. Kernel Databases

II. SEGMENT MANAGEMENT FUNCTIONS

A conceptual discussion of the functions associated with segment management is presented in this chapter. As previously mentioned, two dominating goals of SASS were to provide multilevel controlled access to information and to provide for controlled sharing of information. The major factor in controlled access (which, in effect, refers to information security) and in information sharing is the concept of segmentation. Segmentation, data sharing, and information security will be discussed relative to their value to SASS.

A. BASIC CONCEPTS/DISCUSSION

1. Segmentation

Segmentation has previously been defined as the technique for managing segments of an address space where a segment is defined to be a logical grouping of information which possesses the qualities of having uniform attributes, being logical (vice physical), being visible to the user and being arbitrary in size. Based upon this notion, a process' address space is then viewed as consisting of the collection of segments that the process may access. In a segmented environment all addresses require two components: (1) a segment specifier (number) and (2) the location (offset) within the segment. Each segment may have attached to it

logical attributes that enable certain important control features to be implemented. Controlled access and information sharing implementation is specifically facilitated. By including classification and access information in a segment's logical attributes, a method to enforce information security is provided. Segmentation supports information sharing since it allows a segment to belong to more than one address space, that is, a single physical copy may be accessed by more than one process. Controlled physical sharing of information (within access constraints) is achieved, in the case of SASS, by putting segments which are shared and writeable into the system's global memory (vice a copy in each local memory).

Segmentation also facilitates the implementation of multiple protection domains in SASS. A process' address space is divided into domains or arrangements of segment accessibility. The Kernel domain is the most privileged and includes all segments of the address space, while the supervisor domain is less privileged and excludes segments representing the management of the shared resources by more than one process.

2. Data Sharing

The facility to share a single segment (and thus a single copy of the information to be accessed) by many processes is a significant feature that is facilitated via segmentation. In short, by processes sharing a common

physical copy of a segment, there is no requirement for duplicate copies and thus no possibility exists of having copies that are not up to date. In SASS, given the global memory/local memories environment, the policy is to put copies of segments in local memory except in the case of shared and writeable segments, which are placed in global memory for sharing purposes among processes with the appropriate access.

Segmentation is vital to this policy since only through explicit segmentation can SASS know the read/write properties of the information. Thus, segments which are shared but have read only access (by all processes that may access it) are not put into global memory but rather into the local memory of each of the processes that may access it (viz., multiple copies exist). There is no possibility of the multiple copy/ out of date copy problem since only read access is allowed. However, this is a seeming waste of memory and nonuse of the sharing facility provided by segmentation. The justification is based on a design decision motivated by another goal of SASS -- reduction of bus contention among processors accessing global memory. This is considered to be of more importance than the saving of memory space offered by single copy sharing of information; as stated before, the cost of memory has gone down significantly in the last few years thus reducing its influence on decisions such as this.

3. Information Security

Information security in a computer environment only recently began to receive the attention that it deserves. Few people have been far sighted enough to view computer security in an analogous manner to communications security (an area which has received considerable military and commercial attention throughout history, especially since the advent of electronic communications). Only through harsh and embarrassing lessons has the importance of computer security being recognized. The range of problems encountered covers virtual every level of computer usage: banks and commercial enterprises are victims of theft through the felonious use of computers; universities are the victims of undesired users entering their systems and either maliciously or accidentally destroying valuable programs; and the military faces the real possibility that classified material is being accessed by foreign agents without our knowledge and/or crucial systems are being tampered with without our knowledge. The effects of these type actions may be as small as simple embarrassment or as serious as undermined military preparedness. It should be clear that information security is a serious issue. Definitively, this thesis will consider information security as the process of providing controlled access to information based on proper authorization. A pertinent information security goal is to provide a "multilevel" information security environment

(that is, an environment where multiple levels of sensitive information and user accessibility to that information exist together in a manner such that security is not compromised). The key to achieving computer security lies with the concept of the "security kernel". A discussion of this concept and some supporting definitions is provided in the next section.

a. Basic Security Principles

The protection of secure information in computer systems is affected through two types of control: (1) external controls -- where physical means are used to securely isolate the computer system (e.g., an armed guard) and (2) internal controls -- where the computer itself provides protection by distinguishing information security levels and user accessibilities. Although the discussion in this thesis centers around internal controls, external controls are also a viable and important aspect of the SASS (and other computer system's) information security. As previously stated, the key (or answer) to computer security lies in the security kernel concept. Schell [11] provides a detailed development of the theory behind this concept. The security kernel is defined as that part of the computer system's hardware and software which enforces the authorized access relationships between the user/process (subject) and the accessed information (segment or object).

An important aspect to the development of a secure kernel based system is the security policy to be

enforced. There are two distinct aspects of security policy. The first is the non-discretionary (mandatory) policy that externally constrains what access is permissible; this policy is manifested and implemented in an arrangement where information in the form of a segment (called an "object") is labelled as to its sensitivity; the same is done with the party requesting access (the user/process, called a "subject"). The relationship between the subject and object "labels" that leads to an access permission or denial is defined by a lattice structure [3]. This lattice structure concept will be discussed in the next section. The second aspect of security policy is the discretionary policy, which is a refinement within the non-discretionary constraints. It is emphasized that discretionary security is contained within (and in no way substitutes for) non-discretionary security. An example is the "need to know" policy of the DoD. Implementation of the discretionary security policy for SASS is accomplished in the Supervisor through the maintenance of an Access Control List (ACL) for each file in the file hierarchy. Each access attempt to a file is checked against the ACL and access is granted in accordance with that check and the non-discretionary security check (whichever granted the least access). This allows the users to specify (subject to non-discretionary security constraints) who may access their files. Since the implementation of the discretionary security policy is not a

part of this thesis, a detailed discussion is not provided. Parks [7] provides a discretionary security policy design for SASS.

Implementation of a security policy requires an awareness of and consideration for several basic security properties which are briefly defined below.

The Simple Security Condition restricts a subject's read access to objects whose classification is equal to or less than the subject's classification (the term classification will hereafter be used to indicate a degree of sensitivity or security importance).

The Confinement Property (or "*-property") restricts a subject's write access to objects whose classification is equal to or greater than the subject's classification. This property prevents a subject from writing to an object of lower classification where another subject (of less than the original subject's classification) would have potential access thus violating security.

The Compatibility property has as a basis the hierarchial structure of the objects (segments) of SASS. The objects of SASS are hierarchially organized in a tree structure. The structure consists of nodes, leaves, and a root from which the tree emanates. A node (an alias table that contains a list of attributes for segments) is directly associated with a segment that is the "mentor" for one or more segments. A leaf, viz., a segment, is not an alias

table but may be a mentor segment (with the same access class as the alias table). The Compatibility property basically states that the object access classification must be non-decreasing in moving from the root down the hierarchy (viz., the access classification of a child must be greater than or equal to its parent).

b. Lattice Model Abstraction

A lattice model of secure information flow concept (discussed by Denning [3]) permits concise formulations of the security requirements of different systems and facilitates the construction of mechanisms that enforce security. Specifically, the relationship between classifications can be represented by a partially ordered lattice structure (examples illustrating this concept are presented in the next section). Authorizations for access (or decisions on compatibility) then are based on this lattice. With the properties previously discussed as a basis, the accesses permitted are defined below ("sac" is the abbreviation for "subject access classification" and "oac" for "object access classification" and "|" denotes "not related"):

1. $sac = oac$, read/write permitted
2. $sac > oac$, read permitted
3. $sac < oac$, write permitted
4. $sac \mid oac$, no access permitted

Case 3 represents a subject's ability to "write up", which is a capability not supported in SASS; thus for SASS, case 3 is more accurately represented as:

3. sac < oac, no access permitted

At this point, no design detail has been provided for the representation of a classification or for defining how two classifications are compared and determined to be "equal", "greater than", "less than", or "unrelated" (viz., what does sac>oac mean?). The next section will illustrate these ideas both generally and by examples.

c. Examples

Based on military influence, the examples provided are reflective of a subset of the DoD non-discretionary security policy. A classification (or label) is defined to have two parts: (1) a level (e.g., top secret, secret, confidential, or unclassified in the example) and a category (e.g. Crypto (Cy), Nato (N), Nuclear (Nu), or empty (%) in the example). In the actual implementation (chapter III), provisions will be made for eight levels and sixteen categories. (In some reference texts, levels are called categories and categories are called compartments). The levels are defined by a "totally ordered" relationship where all levels are related:

Top Secret > Secret > Confidential > Unclassified

or

TS > S > C > U

The categories are defined as "disjoint" (no relationship exists when comparing individual categories with other individual categories). The classifications (labels) (the concatenation of level with category) then are defined to have a "partially ordered" relationship since some but not all classifications are related. The cases to be illustrated will be illustrated through a general case and then by specific examples. The general structure is defined by:

Subject's classification = (LS, {CS})

Object's classification = (LO, {CO})

where:

LS = Subject's level

{CS} = Subject's set of categories

LO = Object's level

{CO} = Object's set of categories

The non-inclusive set of partially ordered examples will be chosen from a subset of the classifications derivable from the set of totally ordered levels and the set of disjoint categories:

{TS,S,C,U} and {Cy,N,Nu,%}

Case I : Equal (sac =oac)

General - LS = LO and {CS} = {CO}

Examples - (TS,{Cy,N}) = (TS,{Cy,N})

- (U, {Cy}) = (U ,{Cy})

Access - Read/Write

Case II: Greater than ($sac > oac$)

(1) General - $LS > LO$ and $\{CO\}$ subset to $\{CS\}$

Examples - $(S, \{N, Nu\}) > (C, \{N\})$

- $(S, \{N, Nu\}) > (U, \{\%\})$

(2) General - $LS > LO$ and $\{CS\} = \{CO\}$

Examples - $(TS, \{\%\}) > (S, \{\%\})$

- $(S, \{Nu, N\}) > (U, \{Nu, N\})$

(3) General - $IS = LO$ and $\{CO\}$ proper subset to $\{CS\}$

Examples - $(U, \{Nu\}) > (U, \{\%\})$

$(TS, \{Cy, N, Nu\}) > (TS, \{Cy\})$

Access - Read

Case III: Less than ($sac < oac$)

(1) General - $LS < LO$ and $\{CS\}$ subset to $\{CO\}$

Examples - $(S, \{\%\}) < (TS, \{N\})$

- $(U, \{N, Nu\}) < (C, \{Cy, N, Nu\})$

(2) General - $LS < LO$ and $\{CS\} = \{CO\}$

Examples - $(S, \{\%\}) < (TS, \{\%\})$

- $(U, \{N, Nu\}) < (C, \{N, Nu\})$

(3) General - $LS = LO$ and $\{CS\}$ proper subset to $\{CO\}$

Examples - $(U, \{\%\}) < (U, \{N\})$

- $(C, \{N, Cy\}) < (C, \{N, Nu, Cy\})$

Access = no access (in SASS)

= Write (in principle)

Case IV : Unrelated ($sac \nmid oac$)

(1) General - $LS <, >, \text{ or } = LO$ and $\{CS\} \nmid \{CO\}$

Examples - (S, {N}) | (C, {Nu})
 - (C, {Nu}) | (S, {N})
 - (TS, {N}) | (TS, {Cy})

(2) General - LS > LO and {CS} proper subset to {CO}

Examples - (TS, {X}) | (S, {N})
 - (C, {N, Nu}) | (U, {N, Nu, Cy})

Explanation - there is a contradiction between
 the relationship of the levels and
 the relationship of the categories.
 Since this contradiction is
 unresolvable, the classification
 relationship must be "unrelated".

(3) General - LS < LO and {CO} proper subset to {CS}

Examples - (S, {N, Nu}) | (TS, {N})
 - (U, {Cy}) | (C, {X})

Explanation - same as above

Access = No access

d. Applications to the SASS

The cases above are designed to identify each possible relationship that exists between two labels. In the SASS, it is necessary only to identify cases I and II (label 1 >= label 2), while lumping the other cases into a single case which represents "no access". This arrangement encompasses enforcement of the Confinement Property, Simple Security Condition, and the Compatibility property. Enforcement must occur on every access attempt of an object.

A discussion of the implementation of non-discretionary security policy is provided in the next chapter.

B. SEGMENT MANAGER

1. Function

The Segment Manager is the focal point of the segment management function. Using the per-process Known Segment Table as its database and the Memory Manager and Non-Discretionary Security Module in strongly supportive roles, it is responsible for managing the segmented virtual memory for a process. Its role can be viewed as somewhat intermediary in nature (viz., between the Supervisor modules and the Memory Manager modules). The extended instruction set created in the Segment Manager includes the following instructions: CREATE_SEGMENT, DELETE_SEGMENT, MAKE_KNOWN, TERMINATE, SM_SWAP_IN, and SM_SWAP_OUT (note that the names for SWAP_IN and SWAP_OUT have been modified by preceding each with SM_; this is strictly for clarity because the Memory Manager also creates two instructions called SWAP_IN and SWAP_OUT). These instructions are invoked by the Supervisor domain of the process (viz., calls are made from the Supervisor domain via the Gatekeeper to the Segment Manager in the Kernel domain) to provide SASS support to the Host.

In general, when the Segment Manager receives these calls, it performs certain checks to ensure the validity and

security compliance (when required) of the request (call). These checks are performed using its own database (the KST) and by calls to the Non-Discretionary Security Module (when required). The Segment Manager invokes one of six Memory Manager (more specifically, the Distributed Memory Manager Module) created instructions. These instructions include: MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. These invoked instructions (procedures) in turn perform interprocess communications with the non-distributed memory manager process (where actual memory management functions are accomplished). These interprocess invocations and returns are accomplished through the use of the IPC primitives Signal and Wait. The Segment Manager returns the required arguments to the Supervisor by value (as passed back to it by the Memory Manager and/or determined within itself). The Segment Manager performs actual segment number assignment when a segment is made known to a process' address space. It also performs any further database (KST) updating as may be required. A more detailed description of the specifics of the actions of the Segment Manager will be provided in the implementation described in Chapter III.

2. Database

The Known Segment Table (KST) is the database used to manage segments. The KST is described in its tabular form and PLZ/SVS structured representation in figure 6. There are several basic and pertinent facts to be noted of the KST:

1. It is a process local database; that is, each process has its own KST.
2. The KST is indexed by segment number; each record of the KST consists of a set of fields (description information) regarding a particular segment.
3. Entering information into the fields of a segment is called "making a segment known". This simply refers to adding a segment to a process' address space (viz., making a segment accessible to a process).
4. In SASS, a correspondence exists between making a segment "known" and making a segment "active"; i.e., when a segment is added to the address space of a process, this action results in an entry in the KST (making "known") by the Segment Manager and an entry in the Global Active Segment Table (G_AST) by the Memory Manager process (making it "active").

The G_AST will be described later in this chapter.

A proper description of the structure and fields of the KST is necessary at this point. Using the representation of the PLZ/SYS language structure, the KST is described as an array

of records of fields of varying types. The fields are described separately below. Although the KST index is not in itself a field in the record, it does perform a rather significant role. The KST index is an integer closely related to the segment number of the segment described in that KST entry (viz., it is the subscript into the array of records). This segment number also corresponds to the MMU descriptor register (number) for that segment.

The MM_Handle is the first field in a KST record. The MM_Handle is a system wide unique number that is assigned to each segment with an entry in the G_AST (viz., every active segment). This "handle" is the instrument of controlled single copy sharing of information (segments). It allows a segment to exist under one unique handle but be accessible in the address space of more than one process (with different segment numbers in each address space). The MM_Handle is returned to the Segment Manager by the Memory Manager during the execution of the Make_Known instruction.

The Size field is an integer value (of language structure type "word") which represents the number of 256 byte blocks composing a segment.

The Access_Mode field is used to describe the process' access to the segment (i.e., null or read and/or write).

The In_Core field is used to indicate if the segment is or is not in main memory (i.e., this field is a flag or true/false boolean switch).

The Class field is a long word field used to represent the degree of information sensitivity (viz., access class) assigned to the segment. This field (for example) would be used to numerically describe a classification label (as described above).

The Mentor_Seg_Nr field is a number representing the segment number of a segment's parent or "mentor" segment. Its importance will be discussed shortly.

The Entry_Nr field is a number representing a segment's index number into its parent or mentor segment's Alias Table (not yet discussed).

The Alias Table is a Memory Manager database and will be described later. The aliasing scheme provided via the alias tables is used to prevent passing system wide information out of the Kernel (i.e., the Unique_ID of a segment). The "alias" of a segment is the concatenation of the Mentor_Seg_Nr with the segment's Entry_Nr (index) into the mentor segment's Alias Table. It is clear that the last two fields of a KST record are the "alias" of that segment.

Segment_#

MM_Handle	Size	Access_Mode	In_Core	Class	M_Seg_No	Entry_Number

KST Array [64 KST_REC]

KST_REC Record [MM_Handle Array [3 Word]
 Size Word
 Access_Mode Byte
 In_Core Byte
 Class Long
 M_Seg_No Short_Integer
 Entry_Number Short_Integer]

Figure 6. Known Segment Table.

C. NON-DISCRETIONARY SECURITY MODULE

The key in protection of secure information using internal controls was identified as the security kernel concept. The basic idea within this concept is to prove the hardware part of the Kernel correct and, similarly, to keep the software part small enough so that proving it correct is feasible. A central component of the kernel software is the Non-Discretionary Security Module (hereafter referred to as the NDS Module). The NDS Module is concerned only with the non-discretionary aspect of the security policy in effect; since the discretionary aspect is subservient in nature to the non-discretionary aspect, it is then sufficient that the Kernel contain only the software representing the non-discretionary aspect of the security policy. The discretionary security is provided outside the kernel in the SASS supervisor. Every attempt to access information must result in an invocation of the NDS Module.

The function of the NDS Module is to compare two classifications (viz., compare two labels), make a decision as to their relationship (i.e., =, >, <, |), and return a true/false interpretive answer relative to the query of the calling procedure. The mechanism used as a basis is the lattice model abstraction previously discussed. The NDS Module does not require a database since the labels it compares are stored in (passed from) other Kernel databases.

D. MEMORY MANAGER

1. Function

The Memory Manager process is the only component of the non-distributed kernel. It is responsible for managing the real memory resources of the system -- main (local and global) memory and secondary storage. It is tasked by other processes within the Kernel domain (via Signal and Wait) to perform memory management functions. This thesis will address the Memory Manager in terms of two components: (1) the Memory Manager Process (also called the nondistributed kernel and the Memory Manager Module), and (2) the distributed Memory Manager (also called the Distributed Memory Manager Module). The former is the "true" memory manager while the latter is the interface with other processes, that is, it resolves the issue of interprocess communication with the "true" memory manager.

The Distributed Memory Manager Module creates the following extended instruction set: MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. The instructions form the mechanism of communication between the Segment Manager of a process and a memory manager process (where the actual memory management functions are performed). The Memory Manager Process instruction set corresponds one to one with that of the Distributed Memory Manager; the set consists of: CREATE_ENTRY, DELETE_ENTRY, ACTIVATE, DEACTIVATE, SWAP_IN,

and SWAP_OUT. The basic functions performed by the Memory manager are allocation/deallocation of global and local memory and of secondary storage, and segment transfers from local to global memory (and vice-versa) and from secondary storage to main memory (and vice-versa).

2. Databases

A detailed and descriptive discussion of the Memory Manager databases is presented in the work of Gary and Moore [4] and the reader may refer to it for memory manager database details. This thesis addresses the implementation of the distributed Memory Manager but not the Memory Manager Process, thus brief descriptions are provided of the latter's databases.

The Global Active Segment Table (G_AST) is a system wide (i.e., shared by all memory manager processes) database used to manage all active segments. A lock/unlock mechanism is used to prevent race conditions from occurring. The distributed memory manager of the signalling process locks the G_AST before it signals the memory manager process.

The Local Active Segment Table (L_AST) is a processor local database which contains an entry for each segment active in a process currently loaded in local memory.

The Alias Table is a system wide database associated with each nonleaf segment in the Kernel. It is a product of the aliasing scheme used to prevent passing system wide

information out of the Kernel. The alias table header (provided for file system reconstruction after system crashes) has two pointers, one linking the alias table to its associated segment, the other linking the alias table to the mentor segment's alias table. The fields in the alias table are Unique_ID, Size, Class, Page_Table_Loc, and Alias_Table_Loc. The index into the alias table is Entry_No.

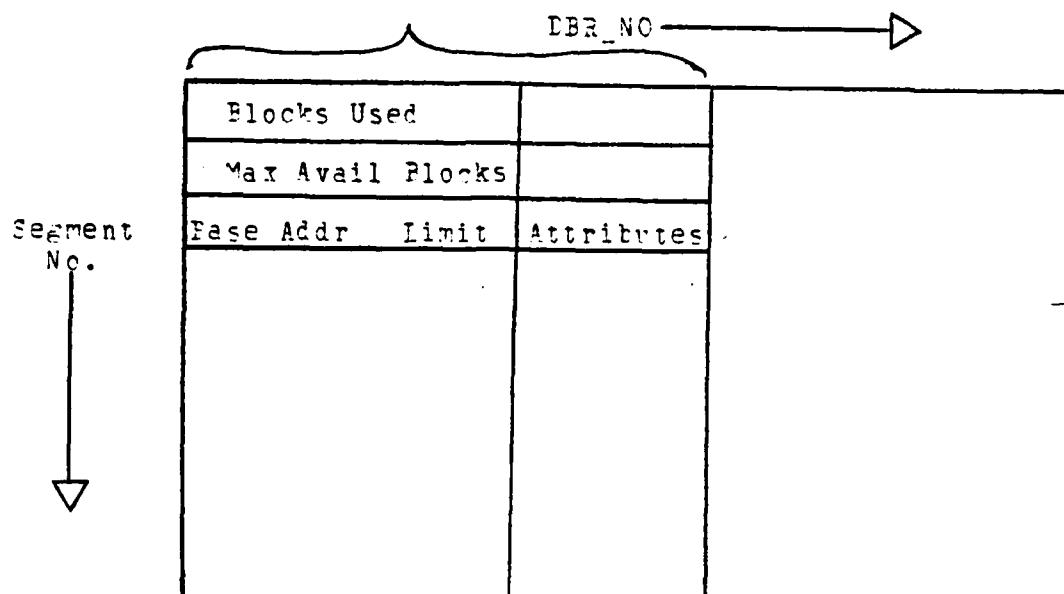
The Memory Management Unit Image (MMU_Image, figure 7) is a processor local database indexed by DBR_No (viz., for each DBR_No there is a MMU_Image record, with each record containing a software image of the segment descriptor registers of the hardware MMU). The MMU_Image is an exact image of the MMU. Each record is indexed by Segment_No (segment number) and each Segment_No entry contains three fields. The Base_Addr field contains the segment's base address in memory. The Limit field contains the number of blocks of contiguous storage for the segment (zero indicates one block). The Attributes field contains 8 flags including 5 which relate to the memory manager. The Elks_Used field and the Max_Blks (available) fields are per record (not per segment entry) and are used in the management of each process' virtual core.

The Memory Bit Maps (Disk_Bit_Map, Global_Memory_Bit_Map, and Local_Memory_Bit_Map) are memory block usage maps that use true/false flags (bits) to indicate the use or availability of storage blocks.

The only database in the Distributed Memory Manager is the Memory Manager CPU Table. It is an array of memory manager VP_ID's (MM_VP_ID) indexed by CPU number. This table enables a signalling process to identify the appropriate memory manager process (virtual processor) to signal.

E. SUMMARY

The segment management functions and key related concepts (such as segmentation) were discussed in this chapter. The importance of segmentation to data sharing and information security was emphasized as were key information security concepts. With this background, the implementation of segment management and a non-discretionary security policy will be described in Chapter III.



MMU_Image Array [Max_DBR_No MMU]
MMU Record [SDR Array [No_Seg_Desc_Reg
 Seg_Desc_Reg]
 Blks_Used Word
 Max_Blks Word]
Seg_Desc_Reg Record [Base_Addr Address
 Limit Byte
 Attributes Byte]
Address Word

Figure 7. Memory Management Unit Image

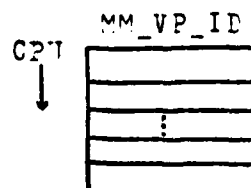


Figure 8. Memory Manager-CPU Table

III. SEGMENT MANAGEMENT IMPLEMENTATION

The implementation of segment management functions and a non-discretionary security policy is presented in this chapter. Paramount to this implementation were several key issues that affected the implementation. These issues are discussed first. The implementation is discussed in terms of the Segment Manager, Non-Discretionary Security (NDS), and Distributed Memory Manager modules.

A. IMPLEMENTATION ISSUES

Segment management for the SASS was provided through the implementation of the Segment Manager Module, the NDS Module, and the Distributed Memory Manager Module. Additionally, since a demonstration/testbed was integral to the testing and verification of the implementation, it was necessary to complete other supportive tasks. Reitz [9] provided a demonstration of the operation of the Inner Traffic Controller primitives SIGNAL and WAIT (for interprocess communication). Integral to this demonstration was the correct performance of the Inner Traffic Controller VP scheduling mechanism and a "stub" of the Traffic Controller and its process scheduling mechanism (the TC support and use of the mechanism of eventcounts and sequencers was not a part of the demonstration). The Segment management demonstration (hereafter referred to as

"Seg_Mgr.Demo") was "built on top of" Reitz' ITC synchronization primitive demonstration (hereafter referred to as "Sync. Demo"). Thus, an immediate issue was to resolve the feasibility of adding on to Sync.Demo and also to refine the present design of the Sync. Demo to facilitate its integration into the Seg_Mgr.Demo. One aspect of this effort was in resolving the problem of how to pass (i.e., in interprocess communication) a larger message.

1. Interprocess Messages

The Sync.Demo passed "word" (16 bit) messages. To provide the mechanism for the distributed memory manager to signal the memory manager process with a command function identification code and the arguments needed to perform that function (e.g., CREATE-ENTRY and its input arguments), a message size of at least eight words (16 bytes) was necessary. An obvious answer was to signal with an array of eight words as the message. PLZ/SYS, however, does not allow passing arrays in its procedure calls (a procedure call is analogous to a subroutine call). Another alternative was to signal with a pointer to the array of words, since PLZ/SYS does allow passing pointers in procedure calls (thus the message would be a pointer to the real message). This, however, would be invalid in the segmented implementation (on the 28000 segmented microprocessor) since identical segment numbers in different processes may not refer to identical segments. For example, a pointer in a process

(e.g., file management) points to an array (i.e., provides its address) by segment number and offset; passing this pointer to another process (e.g., memory manager) would provide this same segment number and offset which, of course, may be a different object in the second process's address space).

Another alternative considered was that of a shared "Mailbox" segment with an associated eventcount acted on by the Kernel Inner Traffic Controller primitives TICKET, ADVANCE and AWAIT. A design for using this concept in the supervisor ring is provided by Parks [7]. This alternative was not deeply considered since these primitives are not included in the current Inner Traffic Controller.

The method ultimately used to signal the new length messages is based on the fact that the ITC is in both the signalling and the receiving (memory manager) processes' address space. The message is loaded into an array in process #1 and a pointer to the array is passed in the call SIGNAL; the VPT, the ITC's database, is then updated by (using the pointer) putting the message into its MSG_Q section. The message is retrieved by process #2 by execution of Reitz' WAIT primitive with only one refinement. That refinement is for the "waiting" process to provide as an argument (in the WAIT primitive) a pointer to its own message array so that the message in the VPT can be copied to it.

This refinement provides for passing a long message essentially "by value" between processes.

2. Structures as Arguments

Another issue concerned the use of pointers in the implementation of segment management. This necessary "evil" is a result of the need to pass linguistically "complex" data types in procedure calls. Complex types refer to array and record structures in PLZ/SYS (as opposed to the "simple" types--byte, word, integer, short-integer, long, and pointer). In managing databases (e.g., KST, G_AST) which consist of arrays of records (which in turn contain records and/or arrays), it was frequently necessary to reference data as an array or record. Within a process, the use of pointers was not a problem (i.e., not a problem such as would be encountered in IPC passing of pointers).

3. Reentrant Code

The issue of code reentrancy was addressed at the assembly language level through the use of a stack segment and registers for storage of local variables. PLZ/SYS (high level language) does not address reentrant procedures and thus the segment management high level code is not automatically reentrant. The problem of reentrancy can be seen by looking at a shared procedure that is not reentrant; such a procedure has storage for its variables allocated statically in memory. Suppose a procedure (e.g., in the Kernel) can be activated by more than one process. While the

procedure is executing in one process, a process switch occurs (e.g., to wait for a disk transfer) and its execution is suspended. The second process is activated, and while it is running it invokes the procedure. While the procedure is executing for the second process it uses the same storage space for variables as it did when executing for the first process. Eventually, it relinquishes the processor. However, when the procedure resumes its execution for the first process, the variable values that were in use by it originally have been changed during its execution in the second process. Thus, incorrect results are now inevitable.

4. Process Structure of the Memory Manager

References to the "Memory Manager" in past works have generally meant the memory manager process (non-distributed kernel). This work references two distinct components of the "memory manager module". The Distributed Memory Manager is an interface provided to the Memory Manager Process. It is, in fact, distributed in the address space of each Supervisor process. In contrast, the Memory Manager Process clearly is not distributed and its address space is contained entirely in the Kernel.

5. Per-Process Known Segment Table

Another key issue was that of the per process Segment Manager database, the KST. Since each process has its own KST, it cannot be linked to the (shared) segment manager procedures. To implement the KST as a per process

database, it was convenient to establish, by convention, a KST segment number that is consistent from process to process. That segment in each process is the KST segment for that process. Implementation is then accomplished by using the segment number to construct a pointer to the base of the appropriate KST. It is then easy to calculate an appropriate offset to index any desired entry in the KST data.

6. DBR Handle

In Reitz's implementation of the multilevel scheduler and the IPC primitives, references to "DBR" (descriptor base register) are references to an address. That address value represents a pointer to an MMU_IMAGE record containing the list of descriptors for segments in the process address space. Gary and Moore [4] reference a "DBR_NO" that is essentially a handle used within the memory manager as an index within the MMU_IMAGE to a particular MMU record. The base address of the MMU record indexed by DBR_NO is then equivalent to the concept of DBR value used in Reitz' work. The effect of this inconsistency on the segment management implementation was minor and will be further discussed later in this chapter.

B. SEGMENT MANAGER MODULE

The Segment Manager Module consists of six procedures representing the six extended instructions it provides. These are based on the design of Coleman [1]. Only calls

from external to the Kernel (via the Gate Keeper) may be made to the Segment Manager (per the loop-free structure of the SASS). The normal sequence of invocation of the Segment Manager functions to allow referencing a segment is: (1) `CREATE_SEGMENT`--allocate secondary storage for the segment and update the mentor segment's Alias Table, (2) `MAKE_KNOWN`--add the segment to the process address space (segment number is assigned), (3) `SWAP_IN`--move the segment from secondary storage into the process's main memory. The normal sequence of invocation to "undo" the above is: (1) `SWAP_OUT`--move the segment from main memory to secondary storage, (2) `TERMINATE`--remove the segment from the process's address space, (3) `DELETE_SEGMENT`--deallocate secondary storage and remove the appropriate entry from the alias table of its mentor segment. The six Supervisor entries into the Segment Manager (viz., the six extended instructions) will be discussed individually below. The PLZ/SYS and PLZ/ASM listings for the Segment Manager are in appendices A and B.

1. Create a Segment

The function that creates a segment (i.e., adds a new segment to the SASS) is `CREATE_SEGMENT`. This function validates the correctness of the Supervisor call by checking the parameters and making certain security checks. The distributed memory manager is then called to accomplish interprocess communication with the Memory Manager Process.

where segment creation is realized through secondary storage allocation and alias table updating.

CREATE_SEGMENT is passed as arguments: (1) Mentor_Seg_No--the segment number of the mentor segment of the segment to be created, (2) Entry_No--the desired entry number in the alias table of the mentor segment, (3) Class--the access class (label) of the segment to be created, and (4) Size--the desired size of the segment (in blocks of 256 bytes). The initial check is to verify that the desired size does not exceed the designed maximum segment size. If this check is satisfactory, a conversion of the Mentor_Seg_No to a KST index is necessary. This is because the Kernel segments use the first several segment numbers available but do not have entries in the KST. Thus if there were 10 Kernel segments and a system segment had segment number 15, then its index in the KST would actually be 5 (i.e., the Kernel segments would use numbers 0-9, and this segment would be the sixth segment in the KST and its index would be 5). A call is then made to the procedure ITC_GET_SEG_PTR with the constant KST_SEG_NO passed as a parameter. This procedure will return a pointer to the base of this process' KST. This pointer is then the basis for addressing entries in the KST. The next check is to see if the mentor segment is known (viz., is in the address space of the process, and thus, in the KST). The key to determining if any segment is known is the mentor segment

entry (M_SEG_No) for that segment in the KST. If not known, this entry in the segment's KST record will be filled with the constant NULL_SEG. The basis for checking to see if the segment's mentor segment is known is the aliasing scheme implication that a mentor segment must be known before a segment can be created. The process classification must next be obtained from the Traffic Controller. The process classification is checked to ensure that it is equal to the classification of the mentor segment since write access to its alias table is needed to create a segment. The NDS module's CLASS_EQ procedure is called and returns a code of true or false. The last check is the compatibility check to ensure that the classification of the segment to be created is greater than or equal to the classification of the mentor segment. This is accomplished by calling the NDS Module's CLASS_GE procedure which returns a code of true or false. If any of these checks are unsatisfactory, an appropriate error code is generated and the Segment Manager returns to its calling point. If all checks are satisfactory, then a pointer to the mentor segment's MM_Handle array is derived (HPTR). Note that in the current memory manager design [4] the actual MM_Handle contents are a Unique_ID (a long word, viz., two words concatenated), and an Index_No (index into the G_AST, a word); thus together these two fields are a total of three words. Since the Segment Manager does not interpret this handle, it is considered a three word array

at this level. For this reason, the entire uninterpreted MM_Handle array will be passed by passing its pointer. This pointer and Entry_No, Size, and Class are then passed in a call to the distributed memory manager procedure MM_CREATE_ENTRY. This procedure, in turn, performs IPC with the memory manager process where segment creation ultimately is accomplished. A success code is returned in an IPC message from the memory manager process via the distributed memory manager to the CREATE_SEGMENT procedure to indicate success or failure as appropriate. This success code is checked by the Segment Manager to ensure confinement would not be violated if it is returned to the calling process' supervisor domain. Only after the success code has been returned can the action of segment creation be considered complete. Segment creation does not imply the ability to reference that segment; MAKE_KNOWN will accomplish that.

2. Delete a Segment

The function that deletes a segment (i.e., deletes a segment from SASS) is DELETE_SEGMENT. Validation of parameters and security checks are performed here similar to (but fewer than) the CREATE_SEGMENT checks. The distributed memory manager is then called to cause IPC with the memory manager process, where segment deletion is realized through secondary storage deallocation and alias table entry deletions. DELETE_SEGMENT is passed as arguments: (1) Mentor_Seg_No and (2) Entry_No. Conversion of the

Mentor_Seg_No to a KST index is accomplished first. The pointer to the base of the KST is located and returned, as before. The mentor segment is checked to ensure it is known, again, by verifying that its own M_SEG_No (mentor segment number) entry in the KST is not the NULL_SEG. The process classification is obtained from the TC and checked (by a call to CLASS_EQ) to ensure it is equal to the mentor segment classification, since deleting an entry requires write access to the alias table. If all checks are satisfactory, then the mentor segment's MM_Handle pointer is derived. This pointer and the mentor segment alias table entry number are passed in a call to the distributed memory manager procedure MM_DELETE_ENTRY. It then performs IPC with the memory manager process where segment deletion is accomplished and a success code is returned as before.

3. Make a Segment Known

The function that makes a segment known (i.e., adds that segment to the process' address space by assigning a segment number, updating the KST, and causing the memory manager process to "activate" the segment (that is, add it to the AST)) is MAKE_KNOWN. Making a segment known is the way the Supervisor declares its intention to use a segment. MAKE_KNOWN is passed as arguments: (1) Mentor_Seg_No, (2) Entry_No, and (3) Access_Desired (e.g., write, read, or null). It returns (1) a success code, (2) the access allowed to the segment, and (3) the segment number. Conversion of

the mentor segment number to a KST index, finding the KST pointer, and verifying that the mentor segment is known occur as previously discussed.

There are three basic cases that may occur in MAKE_KNOWN: (1) the segment is already known (has an entry in the KST), (2) the segment is not known and there is a segment-number available, or (3) the segment is not known and there is no segment number available.

A search is made of the KST using each record's (segment's) M_SEG_No (mentor segment number) and Entry_Number fields as the search key. If these two fields match the input values Mentor_Seg_No and Entry_No, then the record indexed is that of the desired segment; thus the segment to be made known is already known. In this case, all that need be done is to return the success code, segment number (converted from the index by adding to it the number of kernel segments), and the access allowed (equal to the Access_Mode entry in the KST for the already known segment).

During the search of the KST, the M_SEG_No field is also checked to see if it contains the NULL_SEG entry (this implies that the segment number associated with the record is "available"). The first time this is noted, the index is saved. Note the first available index is saved since it is desired to assign segment numbers at the "top" of the KST to keep it dense there. When the search does not find that the segment is already known, the index for the available

segment number is retrieved and converted to segment number by adding to it the number of kernel segments. If this index is the NULL_SEG entry, then there is no segment number available. In this event, the success code is set to NO_SEG_AVAIL, the segment number is assigned NULL_SEG, and access allowed is set to NULL_ACCESS (this is the third case mentioned). If the index is not equal to NULL_SEG and conversion to segment number has occurred then the Traffic Controller is called to provide the DBR_No (descriptor base register number) for the current process. The DBR_No is used by the memory manager process as an index in the MMU_Image and the local AST. The distributed memory manager procedure MM_Activate is called; it is passed the DBR number, the pointer to the mentor segment's MM_Handle entry, the mentor segment alias table Entry_No, and the segment number. MM_Activate performs the normal interface function (performs IPC with the memory manager process procedure that updates the local and global AST's) and also updates the KST entry for the new segment's MM_Handle entry (returned from the memory manager process). It also returns to the Segment Manager the success code, the segment classification, and the segment size from the memory manager process. If the success code is "succeeded" then the issue of access to be granted must be resolved. The process classification is obtained from the TC and passed with the segment classification to the NDS Module procedure CLASS_GE. If the

CONDITION_CODE returned is FALSE then access allowed is NULL_ACCESS, the segment number is NULL_SEG, and MM_DEACTIVATE is called to deactivate the segment. An appropriate error code is returned. If it is greater than or equal then the access allowed is assigned as follows: (1) the two classifications are compared again--this time to see if equal; (2) If they are equal, then the access allowed is either read or write per the access desired; (3) if they are not equal (i.e., the process class is greater than the segment class) then the access allowed is read. Finally the KST entries for that segment number (more accurately for its index in the KST) are filled with the appropriate information (e.g., IN_CORE is false, etc.). If the success code returned from the memory manager process via the distributed memory manager is not "succeeded", then the segment number is set to NULL_SEG and the access allowed is set to NULL_ACCESS.

4. Make a Segment Unknown (Terminate)

The function that makes a segment unknown (i.e., removes that segment from the process' address space--by updating the KST and causing the memory manager process to "deactivate" the segment) is TERMIMATE. It results in removal of the M_SEG_No (mentor segment number) entry from that segment's KST record. Terminate is passed the segment number of the segment to be terminated as an argument. It returns a success code. Conversion of the segment number to

a KST index, finding the KST pointer, and verifying that the segment is known occurs in the same manner as previously discussed. The next check is to verify that the segment is not still loaded in the process' virtual core (viz., it has been "swapped-out"). If not, an error code is returned and the user must cause the Segment Manager extended instruction SM_SWAP_OUT to be executed. The next check is to ensure that the user is not attempting to terminate a Kernel segment. The first several segment numbers in a process' address space will be used by Kernel procedures and data (though they will not be entries in the KST). Thus if there were 10 Kernel segments, then the segment number to be terminated must be greater than or equal to #10 (since the Kernel segments used #'s 0-9). Thus a check is made to ensure that the segment number is not less than the number of Kernel segments; otherwise an error code is returned. Next, the segment number is checked to ensure that it is not larger than the maximum segment number allowable (if so, an error code is returned). If all checks are satisfactory, then the segment's MM_Handle pointer and the process DBR_No are obtained (as discussed before) and passed in a call to the MM_Deactivate procedure. It calls the memory manager process procedure DEACTIVATE which removes or updates (as appropriate) the entries in the local and global AST's.

5. Swap a Segment In

The function that swaps a segment from secondary storage to main memory (global or local) is SM_SWAP_IN. It is passed the segment number of the segment to be swapped in as an argument and returns a success code. Conversion of the segment number to a KST index, finding the KST pointer, and verifying that the segment number is known are accomplished as previously discussed. If the check is satisfactory, then the segment's MM_Handle pointer and the process DBR number are obtained. They are passed with the segment's access mode (from the KST) as arguments in the call to MM_SWAP_IN. It performs normal interface (IPC) functions and returns a success code from the memory manager process' SWAP_IN procedure (where, if not already in core, allocation of main memory space and reading the segment into main memory occurs). If the success code is "succeeded" then the segment's IN_CORE entry in the KST is updated to show that the segment is in main memory for this process (i.e., the entry is now "true").

6. Swap a Segment Out

The function that swaps a segment from main memory to secondary storage is SM_SWAP_OUT. It is passed the segment number of the segment to be swapped out as an argument and returns a success code. The behavior of SM_SWAP_OUT is exactly analogous to that of SM_SWAP_IN except that the segment's KST IN_CORE entry is updated to

reflect that the segment has been removed from main memory for this process (i.e., the new entry is "false").

C. NON-DISCRETIONARY SECURITY MODULE

The Non-Discretionary Security Module implements the non-discretionary security policy for the SASS. The NDS module contains two procedures: CLASS_EQ and CLASS_GE; both compare two labels (classifications) and determine if their relationship meets that of the procedure's name (i.e., equal, or greater than or equal). Although the type of checks being made are, in fact, compatibility checks, Simple Security Condition checks, etc, the NDS Module does not recognize or need to recognize this. It simply uses an algorithm to determine if classification #1 = classification #2 or if classification #1 $\geq$ classification #2, as appropriate. It then returns a condition code of true or false in accordance with the particular case. The earlier discussion of label comparison in accordance with a partially ordered lattice structure is relevant in discussing the NDS Module's algorithm. Consider the same "totally ordered" relationship $TS > S > C > U$ of levels and the "disjoint" relationship $Cy \mid N \mid Nu \mid \%$ of categories. Comparison of levels will be numerical comparisons while comparison of categories will use set theory comparison as a basis. If $TS=4$, $S=3$, $C=2$, $U=1$ are level numerical assignments, then the totally ordered relationship is

maintained (i.e., $TS > S > C > U$ is still true). Now consider the categories and make the following assignments: $Cy=1$, $N=2$, $Nu=4$, $\% = 0$. Note that a classification may have only one level and one category set (the category set may contain several categories). Consider this example: $(TS, \{Cy, N\})$. The level is TS ($=4$). The category is the set $\{Cy, N\}$ and numerically is formed by performing a logical OR with the categories Cy and N . Sixteen bit representation of this is:

$Cy \text{ OR } N$

$(0000 \ 0000 \ 0000 \ 0001) \text{ OR } (0000 \ 0000 \ 0000 \ 0010)$

$= 0000 \ 0000 \ 0000 \ 0011 = \{Cy, N\}$

If $(TS, \{Cy, N\})$ is considered label #1 and $(S, \{N\})$ as label #2 then a comparison of the two labels would be:

(1) Compare level #1 with level #2 -- $4 > 3$?

Clearly, the answer is yes.

(2) Compare category #1 with category #2 -- is

$(0000 \ 0000 \ 0000 \ 0011)$ a superset of

$(0000 \ 0000 \ 0000 \ 0010)$, or more clearly

is the latter a subset of the former?

The answer is yes, and one way to show that is true is by performing a logical OR of category #1 with category #2 and comparing the result to category #1. If the result of the OR operation equals category #1 then category #1 is a superset (not necessarily proper) of category #2. Since usage of the term subset is more frequent than that of superset, this relationship will typically be stated as

"category #2 is a subset of category #1. To illustrate the above:

{Cy,N} OR {N} :

(0000 0000 0000 0011) OR (0000 0000 0000 0010)

= 0000 0000 0000 0011 = category #1.

This means that, in this example, that category #2 is a subset (not necessarily proper) of category #1. Since level #1 > level #2 and category #2 subset category #1 then label #1 > label #2. Thus, a call to the CLASS_EQ procedure with these two labels as the input classifications would return a condition code of false while CLASS_GE would return true. The decision to have the classifications as long word (32 bits) supports the requirement of some DoD specifications for eight levels and sixteen categories. This module uses sixteen bits for the level and sixteen bits for the category. Appendices E and F are the PLZ/SYS and PLZ/ASM listings for the NDS Module.

1. Equal Classification Check

The CLASS_EQ procedure performs comparison of two classifications (labels) and returns a condition code of true if they are equal (an exact match of the two long words bit per bit) or false if they are not.

2. Greater or Equal Classification Check

The CLASS_GE procedure performs comparison of two classifications (labels) and returns a condition code true if classification #1 is greater than or equal to

classification #2 or a condition code of false otherwise. For classification #1 to be greater than or equal to classification #2, the following must be true: (1) level #1 $\geq$ level #2 (determine this by simple numerical comparison of values) and (2) category #2 subset category #1 (determine this by performing a logical OR with the categories and comparing the result to category #1 -- if they are equal then category #2 is a subset of category #1).

Since PLZ/SYS allows passing only "simple" types in calls, the labels were passed as long words (as opposed to each being word arrays of length two). An access class label is never interpreted outside the NDS Module. However, within the NDS Module it is necessary to address the classification's components separately (viz., level and category). Thus, an "overlay" of the logical view of the classification was created. This overlay was a record of type ACCESS_CLASS and it consisted of two fields: level -- 16 bit integer and category -- 16 bit integer. A pointer type CPTR was declared to be of type pointer to ACCESS_CLASS. Two other pointers CLASS1_PTR and CLASS2_PTR were declared to be of type CPTR and were set equal to the base address of CLASS1 and CLASS2 respectively. This "overlay" of the record frame over the two classification labels passed as arguments allowed the desired component addressability.

Futhermore, the non-discretionary policy enforced by SASS can be changed from the current DoD policy to another lattice policy by changing (only) the NDS Module.

D. DISTRIBUTED MEMORY MANAGER MODULE

The Distributed Memory Manager Module performs as an interface between the Segment Manager and the Memory Manager Process. As its name implies, it is distributed in the kernel domain of each Supervisor process. The key role performed in this module is to arrange and perform interprocess communication between its process (actually the VP) and the memory manager process (VP). The module consists of eight procedures. Six of the procedures are called directly by Segment Manager procedures; they are MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. The other two procedures are "service" procedures called by multiple procedures; they are: MM_GET_DBR_VALUE and PERFORM_IPC. The logic used in the first six procedures is somewhat uniform (except for MM_ACTIVATE). Thus, the general logic will be explained (with MM_CREATE_ENTRY as an example) and it should suffice as a description for all (except MM_ACTIVATE) procedures. The service procedures will be described separately.

1. Description of Procedures

Each procedure is invoked (and returns) on a one to one basis with a corresponding procedure in the Segment

Manager. For example, CREATE_SEGMENT invokes MM_CREATE_ENTRY which signals the CREATE_ENTRY procedure in the Memory Manager Process Module. Associated with each procedure is an IPC message "frame" to describe the unique format of the contents of the message to be signalled to the memory manager process. Similarly, there must be a message "frame" for return messages from the memory manager process; this frame is the same for all but the MM_ACTIVATE procedure. Consider the message frame for MM_CREATE_ENTRY; it consists of: (1) a code to describe which function is to be performed (e.g., CREATE_CODE indicates that the CREATE_ENTRY procedure is the intended recipient of the message), (2) MM_Handle (an array of three words), (3) Entry_No, (4) Size, and (5) Class. The message frame has a filler (in this case) of one byte to ensure that it is of length 16 bytes. The purpose of this frame is to provide an overlay onto the actual message array to be signalled and to facilitate loading the arguments into the message array. This is accomplished by having a pointer of the type that points to the frame but by converting its address so that it actually points to the base of the message array. Consider these lines of PLZ/SYS code:

```
CE_MSGPTR := CE_PTR COM_MSGPTR
```

```
CE_MSGPTR^.CREATE_CODE := CREATE_ENTRY_CODE
```

This code is putting a value into the structure pointed to by CE_MSGPTR at entry CREATE_CODE. The key point is that the

frame of that structure is, in fact, CREATE_MSG (as described before), but the physical location pointed to is the message array. This is assured by having the pointer CE_MSGPTR (which points to a structure of type CREATE_MSG) set equal to a pointer (COM_MSGPTR) to the actual message array (COM_MSGBUF). This is accomplished by the first line of code. The message array itself is never directly referenced, but rather the message array that is overlayed by the message frame is filled in the format of the CREATE_MSG frame. In this example, the first two bytes of the message array now contain the value of the constant CREATE_ENTRY_CODE. The remainder of the message array is filled in the same manner (all procedures use the same notion of a frame, although the frames have different formats). The PERFORM_IPC (perform interprocess communication) procedure is called by all procedures at this point in their execution. The key is that the argument passed is the message array pointer not the pointer to the CREATE_MSG record (after all it is only an overlay frame -- linguistically, it is only a type and is never declared as a structure requiring memory storage allocation). When PERFORM_IPC returns, the message array contains a return message. This message consists of only a success code and filler space in all cases but MM_ACTIVATE. Interpretation of the return message is performed in the same manner as loading the message array. The retrieved success code is

returned to the calling Segment Manager procedure. For MM_ACTIVATE, the return message must be interpreted and values for success code, segment size, and segment classification retrieved and returned to the Segment Manager MAKE_KNOWN procedure. The value for the MM_Handle (called the G_AST_Handle by the memory manager process) must be retrieved and entered in the KST record for this segment.

2. Interprocess Communication

The final arrangements and actual performance of IPC is completed by the internal procedure PERFORM_IPC. By locating the identity of the current physical processor (CPU) and using that identity to index into the MM_CPU_TABLE, the VP_ID of the current memory manager is resolved, so that the memory manager process dedicated to this physical processor is signalled. The call to K_LOCK is, in fact, a disguised call to the SPIN_LOCK procedure (since K_LOCK calls SPIN_LOCK). K_LOCK represents an ultimate (as yet unimplemented) goal of a Kernel locking (wait-lock) system. In any event, the G_AST lock must be set prior to signalling the memory manager process. After SIGNAL has been called, a call is made to WAIT with the pointer to the message array as the argument. The synchronization cycle that results is: (1) PERFORM_IPC calls the ITC procedure SIGNAL with the memory manager VP_ID and message array pointer as arguments; PERFORM_IPC then calls WAIT with the message array as the argument, (2) SIGNAL causes the message

array to be copied into the message queue (in the VPT) of the appropriate VP_ID, (3) ultimately, the signalled VP is scheduled; it had previously called WAIT, passing a pointer to its own local message array; the action of WAIT is to copy the message from the VPT to the signalled' process local message array; there it is interpreted by the memory manager process main procedure and the appropriate procedure is called for action (e.g., CREATE_ENTRY). (4) when action is completed the memory manager process fills its local message array with the appropriate return message and calls SIGNAL with a pointer to the message and the original signalling process's VP_ID as arguments, (5) SIGNAL causes the memory manager process' message to be copied into the VPT message queue for the appropriate VP_ID, (6) that VP is eventually scheduled and through the action of WAIT has the return message copied from its message queue in the VPT to its local message array; WAIT then returns to PERFORM_IPC. The G_AST lock is unlocked and PERFORM_IPC returns to the appropriate distributed memory manager procedure.

The last procedure in the distributed memory manager is MM_GET_DBR_VALUE. This procedure simply provides the service of translating a DBR_NO (DBR number) into its appropriate DBR address. It is called by the TC_GETWORK procedure to allow it to call the ITC procedure SWAP_VDBR (remember that presently the Inner Traffic Controller deals with the DBR as the address of the appropriate MMU record in

the MMU_IMAGE while the Traffic Controller uses DBR as a DBR number which indexes to the appropriate MMU record).

E. SUMMARY

The implementation of segment management functions and a non-discretionary security policy for the SASS has been presented in this chapter. The implementation of the Segment Manager Module, Non-Discretionary Security Module, and Distributed Memory Manager management demonstration was described.

Chapter IV will present the conclusions, lessons learned, and suggestions for future work derived from this thesis.

9200	EEEE	0000	DDDD	DDDD	0000	0040	CCCC	CCCC	*.....@.....*
9210	0300	0003	0001	0000	0001	0020	CCCC	CCCC	*..... . . . *
9220	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	*..... . . . *
9230	0200	0003	0001	0000	0001	FFFF	CCCC	CCCC	*..... . . . *
9240	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	*..... . . . *
9250	0000	0003	0001	0000	0002	0060	CCCC	CCCC	*..... < . . . *
9260	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	*..... . . . *
9270	0100	0003	0001	0000	0002	FFFF	CCCC	CCCC	*..... . . . *
9280	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	*..... . . . *

I

Figure 9. Initialized Active Process Table

9000	0000	0060	0000	0000	0000	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*..<.....*
9010	8000	0002	0001	0000	0000	CCCC	0000	0000	0040	FFFF	FFFF	*.....@..*
9020	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*
9030	8100	0001	0001	FFFF	FFFF	FFFF	0000	0000	0060	FFFF	FFFF	*.....<..*
9040	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*
9050	8100	0001	0001	FFFF	FFFF	FFFF	0000	0020	FFFF	FFFF	FFFF	*.....*
9060	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*
9070	8100	0000	0000	0000	0000	0000	0000	FFFF	FFFF	FFFF	FFFF	*.....*
9080	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*
9090	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	*.....*
90A0	FFFF	0020	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*
90B0	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	*.....*
90C0	FFFF	0040	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*
90D0	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	*.....*
90E0	FFFF	0060	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*
90F0	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	EEEE	*.....*
9100	FFFF	FFFF	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	CCCC	*.....*

Figure 10. Initialized Virtual Processor Table


```

ID 8000 10 (MM, partial)
8000 cccc cccc 7000 cccc 6000 cccc cccc cccc
8010 cccc cccc cccc cccc cccc cccc cccc cccc
ID 8100 10 (IDLE, partial)
8100 cccc cccc 7100 cccc 6100 cccc cccc cccc
8110 cccc cccc cccc cccc cccc cccc cccc cccc
ID 8200 10 (IO, partial)
8200 cccc cccc 7200 cccc 9700 cccc cccc cccc
8210 cccc cccc cccc cccc cccc cccc cccc cccc
ID 8300 10 (FM, partial)
8300 cccc cccc 7300 cccc 9300 cccc cccc cccc
8310 cccc cccc cccc cccc cccc cccc cccc cccc

```

.....p.....<.....
.....<.....
.....d.....a.....
.....<.....
.....r.....
.....s.....
.....<.....

Figure 12. Initialized Memory Management Unit Image

```

ID 7090 30 (MM, partial)
7090 CCCC CCCC FFFF 709A 60C0 0000 0000 0000
70A0 0000 0000 0000 0000 5000 8000 7000 AAAA
70B0 6000 AAAA 40A6 709A 0000 FFFF 5000 5000
70C0 7094 FFFF CCCC CCCC CCCC CCCC CCCC CCCC
70D0 CCCC CCCC CCCC CCCC CCCC CCCC CCCC CCCC
70E0 5000 FFFF CCCC CCCC CCCC CCCC CCCC CCCC
ID 7190 30 (IDLE, partial)
7190 CCCC CCCC FFFF 719A 61C0 0000 0000 0000
71A0 0000 0000 0000 0000 5200 8100 7100 AAAA
71B0 6100 AAAA 40A6 719A DDDD FFFF 5000 5200
71C0 7194 FFFF CCCC CCCC CCCC CCCC CCCC CCCC
71D0 CCCC CCCC CCCC CCCC CCCC CCCC CCCC CCCC
71E0 5000 FFFF CCCC CCCC CCCC CCCC CCCC CCCC
ID 7290 30 (IO, partial)
7290 CCCC CCCC FFFF 729A 62C0 0000 0000 0000
72A0 0000 0000 0000 0000 5280 8200 7200 AAAA
72B0 6200 AAAA 40A6 729A 0001 FFFF 5000 5280
72C0 7294 FFFF CCCC CCCC CCCC CCCC CCCC CCCC
72D0 CCCC CCCC CCCC CCCC CCCC CCCC CCCC CCCC
72E0 5000 FFFF CCCC CCCC CCCC CCCC CCCC CCCC
ID 7390 30 (FM, partial)
7390 CCCC CCCC FFFF 739A 63C0 0000 0000 0000
73A0 0000 0000 0000 0000 5400 8300 7300 AAAA
73B0 6300 AAAA 40A6 739A 0002 FFFF 5000 5400
73C0 7394 FFFF CCCC CCCC CCCC CCCC CCCC CCCC
73D0 CCCC CCCC CCCC CCCC CCCC CCCC CCCC CCCC
73E0 5000 FFFF CCCC CCCC CCCC CCCC CCCC CCCC

```

.....p.<.....
.....p...p...
<...@.p....P.P.
p.....
.....
p.....
.....q.a.....
.....R...q...
a...@.q....P.R.
q.....
.....
p.....
.....r.b.....
.....R...r...
b...@.r....P.R.
r.....
.....
p.....
.....s.c.....
.....T...s...
c...@.s....P.T.
s.....
.....
p.....

Figure 13. Initialized Process Stack Segments

```

%ZLINK N=SM8 L=SM8.MAP MM.8 IDLE.8 IO.8 FM.8 TC.8 ITC.8 D MM.1 SM.1 ND SEC.8
PROC DATA = (IDLE DATA IO DATA FM DATA D MM DATA MM DATA)
KER PROC = (ITC INT PROC ITC GLB PROC TC GLB PROC D MM PROC SM PROC NDS PROC)
ZLINK N=SM8 L=SM8.MAP MM.8 IDLE.8 IO.8 FM.8 TC.8 ITC.8 D MM.1 SM.1 ND SEC.8 PROC
DATA = (IDLE DATA IO DATA FM DATA D MM DATA MM DATA) KER PROC = (ITC INT PROC I
TC GLB PROC TC GLB PROC D MM PROC SM PROC NDS PROC)
ZLINK 2.01
LINK COMPLETE
%IMAGER SM8 ($=5000 MM PROC $=5200 IDLE PROC $=5280 IO PROC $=5400 FM PROC $=560
0 KER PROC $=6200 PROC DATA $=8000 MMU DATA $=9000 ITC DATA $=9200 TC DATA) '500
0 6500 O=SYNC.8
IMAGER SM8 ($=5000 MM PROC $=5200 IDLE PROC $=5280 IO PROC $=5400 FM PROC $=5600
KER PROC $=6200 PROC DATA $=8000 MMU DATA $=9000 ITC DATA $=9200 TC DATA) '5000
6500 O=SYNC.8
IMAGER 2.0
1227 BYTES LOADED
%

```

Figure 14. Linker and Imager Command Lines

```

*
NMI
[LOAD SYNC.8
ENTRY POINT 5000
[LOAD DBR.8
ENTRY POINT 8000
[LOAD SYNC STACK.8
ENTRY POINT 7000
[LOAD SYNC DATA.8
ENTRY POINT 9000
[LOAD KST DATA.8
ENTRY POINT 9300
[R R15
R15 40A0 [71B4
RPC 075E [560E
RFC 5040 [5000
[

```

Figure 15. Load Command Lines and Register Initialization

```

IO: READ COMMAND
-----
FM: IO = SIGNALLER
FM: CALL KERNEL(CREATE)
-----
      KERNEL = SIGNALLER(FOR FM)
      MM: CREATE ENTRY
-----
FM: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
      FM: IO = SIGNALLER
      FM: CALL KERNEL(MAKE KNOWN)
-----
      KERNEL = SIGNALLER(FOR FM)
      MM: ACTIVATE
-----
FM: RETURN FROM KERNEL
-----
FM: CALL KERNEL(SWAP IN)
-----
NMI
[

```

Figure 16. Generated Output

```

-----
      KERNEL = SIGNALLER(FOR FM)
      MM: SWAP IN
-----
      FM: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
IO: CALL KERNEL(MAKE KNOWN)
-----
      KERNEL = SIGNALLER(FOR IO)
      MM: ACTIVATE
-----
IO: RETURN FROM KERNEL
-----
IO: CALL KERNEL(SWAP IN)
-----
      KERNEL = SIGNALLER(FOR IO)
      MM: SWAP IN
-----

```

```

NMI
I

```

Figure 16. Generated Output (continued)

```

IG
IO: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
FM: IO = SIGNALLER
FM: CALL KERNEL(SWAP OUT)
-----
KERNEL = SIGNALLER(FOR FM)
MM: SWAP OUT
-----
FM: RETURN FROM KERNEL
-----
FM: CALL KERNEL(TERMINATE)
-----
KERNEL = SIGNALLER(FOR FM)
MM: DEACTIVATE
-----
FM: RETURN FROM KERNEL
-----
IO: READ COMMAND
NMI
[

```

Figure 16. Generated Output (continued)

AD-A094 569

NAVAL POSTGRADUATE SCHOOL MONTEREY CA
IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A SECURE ARCHIVAL STOR--ETC(U)
SEP 80 J T WELLS

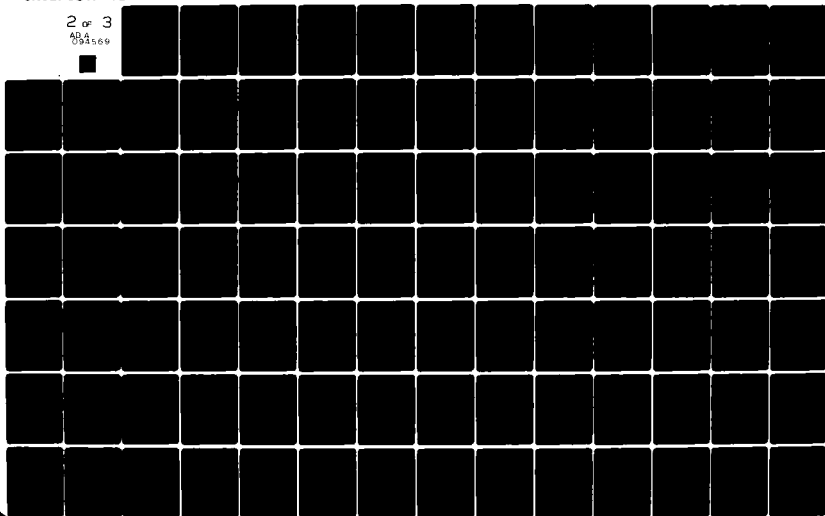
F/G 9/2

UNCLASSIFIED

ML

2 of 3

AD-A094569



```

IG
-----
IO: CALL KERNEL(SWAP OUT)
-----
      KERNEL = SIGNALLER(FOR IO)
      MM: SWAP OUT
-----
IO: RETURN FROM KERNEL
-----
IO: CALL KERNEL(TERMINATE)
-----
      KERNEL = SIGNALLER(FOR IO)
      MM: DEACTIVATE
-----
IO: RETURN FROM KERNEL
-----
IO: READ COMMAND
-----
      FM: IO = SIGNALLER
      FM: CALL KERNEL(DELETE)
-----
      KERNEL = SIGNALLER(FOR FM)
-----
NMI
I

```

Figure 16. Generated Output (continued)

[G

MM: DELETE ENTRY

FM: RETURN FROM KERNEL

IO: READ COMMAND

FM: IO = SIGNALLER

FM: CALL KERNEL(CREATE)

KERNEL = SIGNALLER(FOR FM

NMI
I

Figure 16. Generated Output (continued)

IV. CONCLUSIONS AND FOLLOW ON WORK

The implementation of segment management for the security kernel of a secure archival storage system has been presented. The implementation was completed on Zilog's Z8002 sixteen bit nonsegmented microprocessor. Segmentation hardware (Zilog's Z8010 Memory Management Unit) was not available, therefore it was simulated in software as described by Reitz [9]. The loop free modular construction used in the implementation facilitates ease of expansion or modification.

A non-discretionary security policy was implemented using a partially ordered lattice structure as a basis. Enforcement was realized through an algorithm that compared two labels and determined if their relationship was equal to a desired relationship. Although the DoD security classification system was represented, any non-discretionary security policy that may be represented by a lattice structure may similarly be implemented. This implementation has shown that by having the non-discretionary security policy enforced in one module, changing to another policy requires changing only this one module.

Software engineering techniques used in previous work emphasized the advantages of working with code that is well structured, well documented, and well organized. Despite

being written in assembly language, Reitz' implementation of multiprogramming and process management proved to be consistent in style, clarity and documentation. This enhanced the construction of a segment management demonstration which was built onto his synchronization demonstration. Further, refinements made to his code (not necessitated by any failures of his code) were relatively easily accomplished.

While the segment management implementation appears to perform properly, it has not been subjected to a formal test plan. Such a test plan should be developed and implemented.

The Memory Manager Process has been designed but not implemented. Segment management implementation, provision for IPC using more practical size messages, and the detailed design of the memory manager by Moore and Gary [4], provide a sound foundation for memory manager implementation. A framework of the mainline code needed is provided in the memory manager module of the demonstration code in Appendix I. Prior to this implementation, formal testing of the segment management implementation herein and the monitor implemented by Reitz [9] should be completed.

APPENDIX A - SEGMENT MANAGER PLZ/SYS LISTINGS

SEGMENT_MANAGER

MODULE

CONSTANT

```

NULL_ACCESS      := 4
NULL_SEG         := -1
MAX_NO_KST_ENTRIES := ?? !TO BE DETERMINED!
MAX_SEG_SIZE     := ?? !TO BE DETERMINED!
MAX_SEG_NO       := ?? !TO BE DETERMINED!
KST_SEG_NO       := ?? !TO BE DETERMINED!
NR_OF_KSEGS      := ?? !TO BE DETERMINED!
FALSE            := 0
TRUE             := 1
READ             := 1
WRITE            := 2

```

! ****

```

SUCCESS_CODES    **** !
SUCCEEDED        := 2
MENTOR_SEG_NOT_KNOWN := 22
ACCESS_CLASS_NOT_EQ := 33
NOT_COMPATIBLE   := 24
SEGMENT_TOO_LARGE := 25
NO_SEG_AVAIL     := 27
SEGMENT_NOT_KNOWN := 26
SEGMENT_IN_CORE  := 29
KERNEL_SEGMENT   := 30
INVALID_SEGMENT_NO := 31
NO_ACCESS_PERMITTED := 32
LEAF_SEG_EXISTS  := 10
NO_LEAF_EXISTS   := 11
ALIAS_DOES_NOT_EXIST := 23
NO_CHILD_TO_DELETE := 20
G_AST_FULL       := 12
L_AST_FULL       := 13
LOCAL_MEMORY_FULL := 16
GLOBAL_MEMORY_FULL := 17
SEC_STOR_FULL    := 21
PROC_CLASS_NOT_GE_SEG_CLASS := 41

```

TYPE

```

H_ARRAY      ARRAY [ 3      WORD ]

KST_REC      RECORD [ MM_HANDLE      H_ARRAY
                      SIZE            WORD
                      ACCESS_MODE     BYTE
                      IN_CORE         BYTE
                      CLASS           LONG
                      M_SEG_NO        SECRT_INTEGER
                      ENTRY_NUMBER    SHORT_INTEGER ]

KST          ARRAY [ MAX_NO_KST_ENTRIES KST_REC ]

```

```

KSTPTR      ^KST
ADDRESS      WORD
SEG_ARRAY    ARRAY [MAX_SEG_SIZE  BYTE]

```

EXTERNAL

```

CLASS_EQ PROCEDURE
  RETURNS ( CONDITION_CODE  BYTE )

```

```

CLASS_GE PROCEDURE
  RETURNS ( CONDITION_CODE  BYTE )

```

```

MM_CREATE_ENTRY PROCEDURE
  RETURNS ( SUCCESS_CODE  BYTE )

```

```

MM_DELETE_ENTRY PROCEDURE
  RETURNS ( SUCCESS_CODE  BYTE )

```

```

MM_MAKE_KNOWN PROCEDURE
  RETURNS ( SUCCESS_CODE  BYTE
           CLASS          LONG
           SIZE           WORD )

```

```

MM_TERMINATE PROCEDURE
  RETURNS ( SUCCESS_CODE  BYTE )

```

```

MM_SWAP_IN PROCEDURE
  RETURNS ( SUCCESS_CODE  BYTE )

```

```

MM_SWAP_OUT PROCEDURE
  RETURNS ( SUCCESS_CODE  BYTE )

```

```

TC_GET_PROC_CLASS PROCEDURE
  RETURNS ( PROC_CLASS  LONG )

```

```

ITC_GET_SEG_PTR PROCEDURE
  RETURNS ( SEGPTR ^SEG_ARRAY )

```

```

MONITOR PROCEDURE
!TO BE IMPLEMENTED AT ASSEMBLY LEVEL - SIMPLY WILL
CALL THE MONITOR AT ADDRESS %059A!

```

INTERNAL

```

HPTR      ^H_ARRAY
KPTR      ^KSTPTR

```

GLOBAL

```

!*****
*
*   CREATE_SEGMENT PROCEDURE. INVOKED BY SUPERVISOR
*   PROCEDURE VIA GATE KEEPER. ENSURES CALL IS VALID
*   BY CHECKING TO SEE IF MENTOR SEGMENT KNOWN, IF
*   SEGMENT SIZE IS TOO LARGE (DESIGNED MAX SIZE), IF
*   ACCESS CLASSIFICATION ARE EQUAL, AND IF COMPATIBILITY
*   REQUIREMENTS MET. IF ALL CHECKS ARE SATISFACTORY
*   THEN MM_CREATE_ENTRY IS CALLED FOR MEMORY MANAGER
*   TO TAKE ACTION.
*
*****

```

```

CREATE_SEGMENT PROCEDURE ( MENTOR_SEG_NO    SHORT_INTEGER
                          ENTRY_NO          SHORT_INTEGER
                          CLASS             LONG
                          SIZE              WORD
                          RETURNS ( SUCCESS_CODE    BYTE )

```

```

! **** NOTE:   REENTRANT PROCEDURE   **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !
LOCAL   M_SEG_INDEX    SHORT_INTEGER
ENTRY
  IF SIZE > MAX_SEG_SIZE THEN
    SUCCESS_CODE := SEGMENT_TOO_LARGE
  ELSE
    M_SEG_INDEX := MENTOR_SEG_NO - NR_OF_KSEGS
    KPTR := KSTPTR ITC GET_SEG_PTR ( KST_SEG_NO )
    IF KPTR[M_SEG_INDEX].M_SEG_NO = NULL_SEG THEN
      SUCCESS_CODE := MENTOR_SEG_NOT_KNOWN
    ELSE
      PROC_CLASS := TC_GET_PROC_CLASS
      CONDITION_CODE := CLASS_EQ ( PROC_CLASS,
                                   KPTR[M_SEG_INDEX].CLASS )
      IF CONDITION_CODE = FALSE THEN
        SUCCESS_CODE := ACCESS_CLASS_NOT_EQ
      ELSE
        CONDITION_CODE := CLASS_GE ( CLASS,
                                      KPTR[M_SEG_INDEX].CLASS )
        IF CONDITION_CODE = FALSE THEN
          SUCCESS_CODE := NOT_COMPATIBLE
        ELSE
          HPTR := #KPTR[M_SEG_INDEX].MM_HANDLE
          SUCCESS_CODE := MM_CREATE_ENTRY ( HPTR,
                                             ENTRY_NO, SIZE, CLASS )
          CONFINEMENT_CHECK (SUCCESS_CODE)
        FI
      FI
    FI
  FI
  RETURN
END CREATE_SEGMENT

```

```

*****
*
*   DELETE_SEGMENT PROCEDURE. INVOKED BY SUPERVISOR
*   PROCEDURE VIA TTY GATE KEEPER. CHECKS TO SEE IF
*   MENTOR SEGMENT KNOWN AND IF ACCESS CLASSIFICATION
*   ARE EQUAL. THEN CALLS MM_DELETE_ENTRY FOR MEMORY
*   MANAGER ACTION.
*
*****!

DELETE_SEGMENT PROCEDURE ( MENTOR_SEG_NO    SHORT_INTEGER
                          ENTRY_NO         SHORT_INTEGER )
                          RETURNS ( SUCCESS_CODE    BYTE )

! **** NOTE:  REENTRANT PROCEDURE **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

LOCAL   M_SEG_INDEX    SHORT_INTEGER
ENTRY
  M_SEG_INDEX := MENTOR_SEG_NO - NR_OF_KSEGS
  KPTR := KSTPTR ITC_GET_SEG_PTR ( EST_SEG_NO )
  IF KPTR[M_SEG_INDEX].M_SEG_NO = NULL_SEG THEN
    SUCCESS_CODE := MENTOR_SEG_NOT_KNOWN
  ELSE
    PROC_CLASS := TC_GET_PROC_CLASS
    CONDITION_CODE := CLASS_EQ ( PROC_CLASS,
                                KPTR[M_SEG_INDEX].CLASS )
    IF CONDITION_CODE = FALSE THEN
      SUCCESS_CODE := ACCESS_CLASS_NOT_EQ
    ELSE
      HPTR := #KPTR[M_SEG_INDEX].MM_HANDLE
      SUCCESS_CODE := MM_DELETE_ENTRY ( HPTR, ENTRY_NO )
      CONFINEMENT_CHECK (SUCCESS_CODE)
    FI
  FI
  RETURN
END DELETE_SEGMENT

```

```

!*****
*
* MAKE_KNOWN PROCEDURE. INVOKED BY SUPERVISOR
* PROCEDURE VIA GATE KEEPER. CHECKS TO SEE IF
* MENTOR SEGMENT KNOWN. SEARCHES KST TO SEE IF
* SEGMENT ALREADY KNOWN (WHILE ALSO NOTING THE
* FIRST AVAILABLE (UNUSED) SEG #). IF THE SEGMENT
* IS ALREADY KNOWN THEN FINISHED. IF NOT, THEN,
* USING THE AVAILABLE SEG #, CALL MM_ACTIVATE FOR
* MEMORY MANAGER ACTION. THEN CHECK TO SEE IF ANY
* ACCESS IS ALLOWED. IF YES THEN DETERMINE THE
* APPROPRIATE ACCESS ALLOWED AND UPDATE KST. IF
* NO, THEN RETURN NULL ACCESS.
*
*****!

```

```

MAKE_KNOWN PROCEDURE ( MENTOR_SEG_NO  SHORT_INTEGER
                      ENTRY_NO        SHORT_INTEGER
                      ACCESS_DESired  BYTE )
    RETURNS ( SEGMENT_NO  SHORT_INTEGER
             ACCESS_ALLOWED  BYTE
             SUCCESS_CODE  BYTE )
! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL
    AVAIL_SEG SHORT_INTEGER
    INDEX     SHORT_INTEGER
    DER_NO    SHORT_INTEGER
    M_SEG_INDEX SHORT_INTEGER
ENTRY
    M_SEG_INDEX := MENTOR_SEG_NO - NR_OF_KSEGS
    KPTR := KSTPTR ITC_GET_SEG_PTR ( KST_SEG_NO )
    IF KPTR^[M_SEG_INDEX].M_SEG_NO = NULL_SEG THEN
        SUCCESS_CODE := MENTOR_SEG_NOT_KNOWN
        SEGMENT_NO := NULL_SEG
        ACCESS_ALLOWED := NULL_ACCESS
    ELSE
        PROC_CLASS := TC_GET_PROC_CLASS
        HPTR := #KPTR^[M_SEG_INDEX].MM_HANDLE
        INDEX := 0
        SEGMENT_NO := NULL_SEG
        AVAIL_SEG := NULL_SEG
        SEE_IF_KNOWN:
            DO
                IF KPTR^[INDEX].M_SEG_NO = MENTOR_SEG_NO
                ANDIF KPTR^[INDEX].ENTRY_NUMBER = ENTRY_NO THEN
                    !CASE: SEGMENT ALREADY KNOWN!
                    SUCCESS_CODE := SUCCEEDED
                    SEGMENT_NO := INDEX + NR_OF_KSEGS
                    ACCESS_ALLOWED := KPTR^[INDEX].ACCESS_MODE
                    EXIT SEE_IF_KNOWN
            END DO

```

```

ELSE
  IF KPTR^[INDEX].M_SEG_NO = NULL_SEG
    ANDIF AVAIL_SEG = NULL_SEG THEN
      AVAIL_SEG := INDEX + NR_OF_KSEGS
    FI
    INDEX += 1
    IF INDEX > MAX_NO_KST_ENTRIES THEN EXIT FI
  FI
OD
! EXIT SEE IF KNOWN !
IF SEGMENT_NO = NULL_SEG
  ANDIF AVAIL_SEG <> NULL_SEG THEN !CASE: SEGMENT NOT
    KNOWN AND SEG # IS AVAILABLE!
    INDEX := AVAIL_SEG - NR_OF_KSEGS
    SEGMENT_NO := AVAIL_SEG
    DBR_NO := TC_GET_DBR_NO
    SUCCESS_CODE, KPTR^[INDEX].CLASS, KPTR^[INDEX].SIZE :=
      MM_ACTIVATE(DBR_NO, EPTR, ENTRY_NO, SEGMENT_NO)
    CONFINEMENT_CHECK (SUCCESS_CODE)
    IF SUCCESS_CODE = SUCCEEDED THEN
      CONDITION_CODE := CLASS_GE ( PROC_CLASS,
        KPTR^[INDEX].CLASS )
      IF CONDITION_CODE = FALSE THEN !NO ACCESS!
        ACCESS_ALLOWED := NULL_ACCESS
        SUCCESS_CODE := MM_DEACTIVATE (DBR_NO, EPTR)
        CONFINEMENT_CHECK (SUCCESS_CODE)
        SUCCESS_CODE := PROC_CLASS_NOT_GE_SEG_CLASS
        SEGMENT_NO := NULL_SEG
      ELSE
        CONDITION_CODE := CLASS_EQ ( PROC_CLASS,
          KPTR^[INDEX].CLASS )
        IF CONDITION_CODE = TRUE
          ANDIF ACCESS_DESIRED = WRITE THEN
            ACCESS_ALLOWED := WRITE
          ELSE
            ACCESS_ALLOWED := READ
          FI
          KPTR^[INDEX].IN_CORE := FALSE
          KPTR^[INDEX].M_SEG_NO := MENTOR_SEG_NO
          KPTR^[INDEX].ENTRY_NUMBER := ENTRY_NO
          KPTR^[INDEX].ACCESS_MODE := ACCESS_ALLOWED
        FI
      ELSE
        SEGMENT_NO := NULL_SEG
        ACCESS_ALLOWED := NULL_ACCESS
      FI
    ELSE
      SUCCESS_CODE := NO_SEG_AVAIL
      SEGMENT_NO := NULL_SEG
      ACCESS_ALLOWED := NULL_ACCESS
    FI
  FI
RETURN
END MAKE_KNOWN

```

```

!*****
*
*   TERMINATE PROCEDURE. INVOKED BY SUPERVISOR
*   PROCEDURE VIA GATE KEEPER. CHECKS TO SEE IF
*   SEGMENT IS KNOWN AND IF SEGMENT NUMBER IS
*   VALID. IF CHECKS ARE SATISFACTORY THEN MM_DEACTI-
*   VATE IS CALLED FOR MEMORY MANAGER ACTION.
*
*****!

```

```

TERMINATE PROCEDURE ( SEGMENT_NO  SHORT_INTEGER )
    RETURNS          ( SUCCESS_CODE BYTE )

```

```

! ****      NOTE: REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL      INDEX      SHORT_INTEGER
           DBR_NO      SHORT_INTEGER

```

```

ENTRY
    INDEX := SEGMENT_NO - NR_OF_KSEGS
    KPTR := KSIPTER ITC_GET_SEG_PTR ( KSI_SEG_NO )
    IF KPTR^[INDEX].M_SEG_NO = NULL_SEG THEN
        SUCCESS_CODE := SEGMENT_NOT_KNOWN
    ELSE
        IF KPTR^[INDEX].IN_CORE = TRUE THEN
            SUCCESS_CODE := SEGMENT_IN_CORE
        ELSE
            IF SEGMENT_NO < NR_OF_KSEGS
                SUCCESS_CODE := KERNEL_SEGMENT
            ELSE
                IF SEGMENT_NO > MAX_SEG_NO THEN
                    SUCCESS_CODE := INVALID_SEGMENT_NO
                ELSE
                    HPTR := #KPTR^[INDEX].MM_HANDLE
                    DBR_NO := TC_GET_DBR_NO
                    SUCCESS_CODE := MM_DEACTIVATE (DBR_NO, HPTR)
                    CONFINEMENT_CHECK (SUCCESS_CODE)
                    IF SUCCESS_CODE = SUCCEEDED THEN
                        KPTR^[INDEX].M_SEG_NO := NULL
                    FI
                FI
            FI
        FI
    FI
    RETURN
END TERMINATE

```

```

!*****
*
*   SM_SWAP_IN PROCEDURE. INVOKED BY SUPERVISOR
*   PROCEDURE VIA THE GATE KEEPER. CHECKS TO SEE IF
*   SEGMENT KNOWN; IF YES THEN CALLS MM_SWAP_IN FOR
*   MEMORY MANAGER ACTION.
*
*****!

```

```

SM_SWAP_IN PROCEDURE ( SEGMENT_NO  SHORT_INTEGER )
                     RETURNS ( SUCCESS_CODE BYTE )

```

```

! ****      NOTE: REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL      INDEX      SHORT_INTEGER
           DBR_NO      SHORT_INTEGER

ENTRY
  INDEX := SEGMENT_NO - NF_OF_KSEGS
  KPTR := KSTPTR ITC_GET_SEG_PTR ( KST_SEG_NO )
  IF KPTR^[INDEX].M_SEG_NO = NULL_SEG THEN
    SUCCESS_CODE := SEGMENT_NOT_KNOWN
  ELSE
    IF KPTR^[INDEX].IN_CORE = TRUE THEN
      SUCCESS_CODE := SUCCEEDED
    ELSE
      EPTR := #KPTR^[INDEX].MM_HANDLE
      DBR_NO := TC_GET_DBR_NO
      SUCCESS_CODE := MM_SWAP_IN ( EPTR, DBR_NO,
                                   KPTR^[INDEX].ACCESS_MODE )
      CONFINEMENT_CHECK (SUCCESS_CODE)
      IF SUCCESS_CODE = SUCCEEDED THEN
        KPTR^[INDEX].IN_CORE := TRUE
      FI
    FI
  FI
  FI
  RETUPN
END SM_SWAP_IN

```

```

!*****
*
*   SM_SWAP_OUT PROCEDURE. INVOKED BY SUPERVISOR
*   PROCEDURE VIA THE GATE KEEPER. CHECKS TO SEE IF
*   SEGMENT KNOWN; IF YES THEN MM_SWAP_OUT IS CALLED
*   FOR MEMORY MANAGER ACTION.
*
*****

```

```

SM_SWAP_OUT PROCEDURE ( SEGMENT_NO  SHORT_INTEGER )
    RETURNS ( SUCCESS_CODE BYTE )

```

```

! ****      NOTE: REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL      INDEX      SHORT_INTEGER
          DBR_NO      SHORT_INTEGER

```

```

ENTRY
    INDEX := SEGMENT_NO - NR_OF_KSEGS
    KPTR := KSTPTR ITC_GET_SEG_PTR (KST_SEG_NO )
    IF KPTR^[INDEX].M_SEG_NO = NULL_SEG THEN
        SUCCESS_CODE := SEGMENT_NOT_KNOWN
    ELSE
        IF KPTR^[INDEX].IN_CORE = FALSE THEN
            SUCCESS_CODE := SUCCEEDED
        ELSE
            HPTR := #KPTR^[INDEX].MM_HANDLE
            DBR_NO := TC_GET_DBR_NO
            SUCCESS_CODE := MM_SWAP_OUT ( DBR_NO,HPTR )
            CONFINEMENT_CHECK (SUCCESS_CODE)
            IF SUCCESS_CODE = SUCCEEDED THEN
                KPTR^[INDEX].IN_CORE := FALSE
            FI
        FI
    FI
    RETURN
END SM_SWAP_OUT

```

```

!*****
*
*   CONFINEMENT_CHECK PROCEDURE. SERVICE PPOCEDURE TO
*   ENSURE NO SECURITY VIOLATION OCCURS WHEN INFO IS
*   PASSED OUT OF THE KERNEL VIA THE SUCCESS_CODE.
*   CALLS ASSEMBLY PROCEDURE - MONITOR WHICH IS TO
*   CAUSE A JUMP TO THE MONITOR'S ADDRESS 2259A.
*
*****!

```

CONFINEMENT_CHECK PROCEDURE (SUCCESS_CODE BYTE,

```

! ****      NOTE: REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

ENTRY

IF SUCCESS_CODE

```

CASE LEAF_SEG_EXISTS THEN
MONITOR !EXIT SYSTEM!
CASE NO_LEAF_EXISTS THEN
MONITOR !EXIT SYSTEM!
CASE ALIAS_DOES_NOT_EXIST THEN
MONITOR !EXIT SYSTEM!
CASE NO_CHILD_TO_DELETE THEN
MONITOR !EXIT SYSTEM!
CASE G_AST_FULL THEN
MONITOR !EXIT SYSTEM!
CASE L_AST_FULL THEN
MONITOR !EXIT SYSTEM!
CASE LOCAL_MEMORY_FULL THEN
MONITOR !EXIT SYSTEM!
CASE GLOBAL_MEMORY_FULL THEN
MONITOR !EXIT SYSTEM!
CASE SEC_STOR_FULL THEN
MONITOR !EXIT SYSTEM!

```

FI

RETURN

END CONFINEMENT_CHECK

END SEGMENT_MANAGER

APPENDIX B - SEGMENT MANAGER PL7/ASM LISTINGS

```

2 !
3 !
4 SEG_MGR MODULE
5
6
7
8
9
10
11 MAX_SEG_NO_KST_ENTRIES
12 MAX_SEG_SIZE
13 KST_SEG_NO
14 NR_OF_KSEGS
15 TRUE
16 FALSE
17 READ
18 WRITE
19 ! **** SUCCESS_CODES **** !
20 SUCCEEDED
21 MENTOR_SEG_NOT_KNOWN
22 ACCESS_CLASS_NOT_EQ
23 NOT_COMPATIBLE
24 SEGMENT_TOO_LARGE
25 NO_SEG_AVAIL
26 SEGMENT_NOT_KNOWN
27 SEGMENT_IN_CORE
28 KERNEL_SEGMENT
29 INVALID_SEGMENT_NO
30 NO_ACCESS_PERMITTED
31 LEAF_SEG_EXISTS
32 NO_LEAF_EXISTS
33 ALIAS_DOES_NOT_EXIST
34 NO_CHILD_TO_DELETE
35 GAST_FULL
36 LAST_FULL
37 PROC_CLASS_NOT_OF_SEG_CLASS :=41

```

!PAGE

38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72

```

LOCAL MEMORY FULL      := 16
GLOBAL MEMORY FULL     := 17
SEC STOR FULL          := 21
MONITOR                 := %059A

TYPE      E_ARRAY      ARRAY [ 3      WORD ]

      KST_REC RECORD
      [ MM_HANDLE      E_ARRAY
        SIZE           WORD
        ACCESS MODE    BYTE
        IN CORE        BYTE
        CLASS          LONG
        M_SEG NO       SHORT INTEGER
        ENTRY_NUMBER   SECT_INTEGER ]

      ADDRESS          WORD

SEG_ARRAY ARRAY [MAX_SEG_SIZE  BYTE]

INTERNAL

$SELECTION KST DCL
!NOTE: THIS SECTION IS AN OVERLAY/FRAME USED TO
DEFINE THE KST FORMAT. NO STORAGE IS ASSIGNED
RATHER THE KST IS STORED IN A SEPARATE
SEGMENT SET ASIDE FOR IT !
$ABS &
KST      ARRAY [MAX_NO_KST_ENTRIES KST_REC]

```

2022

!PAGE

73	EXTERNAL
74	
75	CLASS_EQ PROCEDURE
76	
77	CLASS_GE PROCEDURE
78	
79	MM_CREATE_ENTRY PROCEDURE
80	
81	MM_DELETE_ENTRY PROCEDURE
82	
83	MM_ACTIVATE PROCEDURE
84	
85	MM_DEACTIVATE PROCEDURE
86	
87	MM_SWAP_IN PROCEDURE
88	
89	MM_SWAP_OUT PROCEDURE
90	
91	TC_GET_PROC_CLASS PROCEDURE
92	
93	ITC_GET_SEG_PTR PROCEDURE
94	
95	TC_GET_DBH_NO PROCEDURE
96	
97	
98	IPAGE

```

!
0000
99          $SECTION SM PRCC
100         GLOBAL
101
102         CREATE SEG          PROCEDURE
103
104         !*****
105         ! CHECKS VALIDITY OF CREATE REQUEST AND !
106         ! CALLS MM CREATE IF VALIL.
107         !*****
108         ! REGISTER USE:
109         ! PARAMETERS
110         ! R1: MENTOR_SEG_NO(INPUT)
111         ! R2: ENTRY_NO(INPUT)
112         ! R3: SIZE(INPUT)
113         ! R4: CLASS (INPUT)
114         ! R5: SUCCESS CODE (RETURNFL)
115         ! LOCAL USE
116         ! R9: KST REC INDEX
117         ! R6,R7 VARIOUS USES
118         ! R15: ~KST
119         !*****
120
121         ENTRY
122         CP R3,#MAX_SEG_SIZE
123         IF GT THEN
124             LL R0,#SEGMENT TOO LARGE
125             FALSE
126         SUR R15,#10 ISTACK AREA FOR INPUT REGS!
127         LDM R15,R1,#5
128         LD R1,#KST_SEG_NO
129         CALL ITC GET_SEG_PTR !R1: KST_SEG_NO!
130                                     ! (RET:R0:~KST)!
131         LD R13,R0 !KST BASE ADDRESS (IE ~KST)!
132         LDM R1,R15,#5 !RESTORE NEEDED REGS!
133         IPAGE

```

0020	A119	134	LD	R9,R1	ICOPY OF MENTOR SEG NO!
0028	0309	135	SUB	R9,#NR_OF_KSEGS	ICONVERT MENTOR_SEG_NO
002C	190E	136			KST REC INDEX!
0030	819D	137	MULT	RR8,#SIZEOF_KST_REC	IOFFSET TO KST_REC!
0032	2106	138	ADD	R13,R9	!ADD OFFSET TO KST_REC!
0036	4ADE	139	LD	R6,#NULL_SEG	
003A	5E0E	140	CPB	RL6,ΔST.M_SEG_NO(R13)	
003E	2100	141	IF	EQ	THEN !MENTOR_SEG NOT KNOWN!
0042	5EE8	142	LD	R0,#MENTOR_SEG_NOT_KNOWN	
0046	93FD	143	ELSE		
0048	5F07	144	PUSH	QR15,R13	
004C	97FD	145	CALL	TC_GET_PROC_CLASS	!(RR2:PROC_CLASS)!
004F	54D4	146	PCP	R13,QR15	
0052	93FD	147	LDL	RR4,KST.CLASS(R13)	
0054	5F0E	148	PUSH	QR15,R13	
		149	CALL	CLASS_EQ	!(RR2:PROC_CLASS)!
		150			!(RR4:MENTOR_SEG_CLASS)!
		151			!(R1:(RET)CONDITION_CODE)!
0058	97FD	152	PCP	R13,QR15	
005A	A116	153	LD	R6,R1	
005C	1CF1	154	LDM	R1,QR15,#5	IRESTORE INPUT REGS!
0060	0B06	155	CP	R6,#FALSE	
0064	5E0E	156	IF	EQ	THEN
006E	2100	157	LD	R0,#ACCESS_CLASS_NOT_EQ	
006C	5F0E	158	ELSE		
		159			
0070	93FD	160	PUSH	QR15,R13	ISAVE ^KST!
0072	9442	161	LDL	RR2,RR4	!CLASS!
0074	54D4	162	LDL	RR4,KST.CLASS(R13)	
0078	5F00	163	CALL	CLASS_GE	!(RR2:CLASS)!
		164			!(RR4:MENTOR_CLASS)!
		165			!(RET:R1:COND_CODE)!
007C	97FD	166	PCP	R13,QR15	IRESTORE PTR!
007E	0B01	167	CP	R1,#FALSE	
0082	1CF1	168	LDM	R1,QR15,#5	
		169			!PAGE

!	00DA 5F00	0000*	225	CALL	TC GET PROC CLASS
			226		!(RETURNS R02:PROC_CLASS)!
	00DE 97FD		227	POP	R13,R015
	00EE 54D4	000A	228	LDL	R04,KST.CLASS(R13) IMENTOR SEG CLASS!
	00E4 93FD		229	PUSH	R015,R13
	00EC 5F00	0000*	230	CALL	CLASS_EQ
			231		!(R02:PROCESS CLASS)!
			232		!(R04:MENTOR SEG CLASS)!
			233		!(R1:(RET) CCNDITION_CODE)!
	00EA A11C		234	LD	R6,R1
	00FC 97FL		235	POP	R13,R015
	00EE 97F2		236	POP	R2,R015
	00F0 97F1		237	POP	R1,R015
	00F2 0R06	0000	238	CP	R6,#FALSE
	00F6 5F0E	0102	239	IF	EQ THEN
	00FA 2100	0021	240	LD	R0,#ACCESS_CLASS_NOT_EQ
	00FE 5F0E	010E	241		
	0102 76D1	0000	242	LDA	R1,KST.MM_HANDLE(R13)
	0106 5F00	0000*	243	CALL	MM_DELETE_ENTRY
			244		!(R1:MM_HANDLE)!
			245		!(R2:ENTRY_NO)!
			246		!(R0:(RET)SUCCESS_CODE)!
			247	CALL	CONFINEMENT_CHECK
			248		!(R0:SUCCESS_CODE)!
			249		
	010A 5F00	041A	250	FI	
			251	RET	
	010E 9E0B		252	END DELETE_SEG	
	0110			IPAGE	

! 0110

```

253 MAKE KNOWN
254 !*****PROCEDURE*****!
255 ! CHECKS VALIDITY OF MAKE KNOWN REQUEST AND !
256 ! CALLS MM ACTIVATE IF VALID. ASSIGNS SEG !
257 ! NUMBER AND UPDATES KST. !
258 !*****!
259 ! REGISTER USE: !
260 ! PARAMETERS: !
261 ! R1:MENTOR_SEG_NO(INPUT) !
262 ! R2:ENTRY_NO(INPUT) !
263 ! R3:ACCESS_DESIRED(INPUT) !
264 ! R4:SUCCESS_CODE(RET) !
265 ! R1:SEGMENT_NO(RET) !
266 ! R2:ACCESS_ALLOWED(RET) !
267 ! LOCAL USE !
268 ! IDENTIFIED AT POINT OF USAGE !
269 !*****!
270 ENTRY
271 PUSH @R15,R1 !SAVE INPUT REGS!
272 PUSH @R15,RR2
273 LD R1,#KST_SEG_NO
274 CALL ITC_GET_SEG_PTR ! (R1:KST_SEG_NO,RET:R0:~KST)!
275 LD R13,R0 !KST!
276 POPL RR2,@R15
277 POP R1,@R15
278 LD R5,R1 !COPY OF MENTOR_SEG_NO!
279 SUB R5,#AE_OF_KSEGS !CONVERT TO INDEX!
280 MULT RR4,#SIZEOF_KST_REC !KST OFFSET TO SEG REC!
281 ADD R13,R5 !ADD OFFSET TO ~KST!
282 LD R4,#NULL_SEG
283 CPR R14,AST.M_SEG_NO(R13)
284 IF EQ THEN
285 LD R0,#MENTOR_SEG_NOT_KNOWN
286 LD R1,#NULL_SEG
287 !PAGE

```

```

0110 93F1
0112 91F2
0114 2101 0002
0118 5F00 0000*
011C A10D
011F 95F2
0120 97F1
0122 A115
0124 030E 000A
0128 1904 0010
012C 815E
012F 2104 FFFF
0132 4ADC 000E
0136 5F0E 014A
013A 2103 0010
013E 2101 FFFF

```

0142	2102	0004	LD	R2,#NULL_ACCESS	288
0146	5E08	02BA	ELSE		289
014A	2107	0000	LD	R7,#0 IKST INDEXI	290
014E	210E	FFFF	LD	R8,#NULL_SEG IAVAIL SEG INDEXI	291
0152	A109		LD	R9,R0 I^KSTI	292
0154	210A	FFFF	LD	R10,#NULL_SEG ISEG KNOWN INDICATORI	293
			SEE_IF_KNOWN:		294
			DO		295
015E	4A99	000E	CPB	R11,KST.M_SEG_NO(R9)	296
015C	5E0E	017C	IF EQ THEN		297
0160	4A9A	000F	CPB	RL2,KST.ENTRY_NUMBER(R9)	298
0164	5E0F	017C	IF EQ THEN	ICASE: SEG KNOWNI	299
0168	2100	0002	LD	R0,#SUCCEEDED	300
016C	0107	000A	ADD	R7,#NR OF KSEGS	301
0170	A171		LD	R1,R7 ISEG#I	302
0172	009A	0008	LDB	RL2,KST.ACCESS_MODE(R9)	303
0176	A11A		LD	R10,R1 ISET SEG KNOWN INDICATORI	304
017E	5E0E	01A6	EXIT FROM SEE_IF_KNOWN		305
			FI		306
			FI		307
			CPB	RL4,KST.M_SEG_NO(R9) ISEE IF SEG # AVAILI	308
017C	4A9C	000E	IF EQ THEN		309
0180	5E0F	0192	CP	R8,#NULL_SIG	310
0184	0B28	FFFF	IF EQ THEN		311
0188	5E0E	0192	LD	R8,R7 ISAVE FIRST AVAIL SEG INDEXI	312
018C	A178		ADD	R8,#NR OF KSEGS ICONVIRT TO SEG #I	313
018E	010E	000A	FI		314
			FI		315
			INC	R7	316
0192	A970		ADD	R9,#SIZEOF KST_REC IINCREMENT ONE REC!	317
0194	0109	0010	CP	R7,#MAX NO KST_ENTRIES	318
019E	0FE7	0036	IF GT THEN		319
019C	5E22	01A4	EXIT FROM SEE_IF_KNOWN		320
01A0	5E08	01A6			321
					322

!PAGE

!	01A4	08D9	323	FI	OD
	01A6	082A	324	ISFE	IF KNOWN!
	01AA	5E0E	325	CP	R10,#NULL_SEG
	01AF	080E	326	IF EQ	THEN !SEG KNOWN INDICATOR NOT SET!
	01B2	5E06	327	CP	RE,#NULL_SEG
	01B6	91F0	328	IF NE	THEN !CASE:SEG UNKNOWN AND SEG# AVAIL!
	01P8	91F2	329	PUSHL	0R15,RR0 !KST AND MENTOR_SEG NO!
	01BA	93F8	330	PUSHL	0R15,RR2 !ENTRY NO &ACCESS_DESIRE!
	01FC	93FD	331	PUSH	0R15,RE !SEG #!
	01BE	5F00	332	PUSH	0R15,R13 !MENTOR_SEG REC PTR!
	01C2	A11A	333	CALL	TC_GET_DER_NO ! (R1T:R11:DER_NO)!
	01C4	97FD	334	LD	R10,R1 !DERH_NO!
	01C6	97F8	335	PCP	R13,0R15
	01C8	95F2	336	POP	RE,0R15
	01CA	95FE	337	PCPL	RR2,0R15
			338	PCPL	RR4,0R15
			339	!MUST REARRANGE REGS FOR PASSING AND	
			340	RETURN	CONSISTENCY OF LOCATION!
	01CC	A13E	341	LD	R5,R3 !ACCESS_DESIRE!
	01CE	A123	342	LD	R3,R2 !ENTRY_NO!
	01D0	76D2	343	LDA	R2,KST.MM_HANDLE(R13) !MENTOR !PTR!
	01D4	A116	344	LD	R6,R1 !MENTOR_SEG_NO!
	01D6	A181	345	LD	R1,R6 !SEGMENT_NO (SAVE)!
	01DE	A1E4	346	LD	R4,R3 !SEGMENT_NO (PASSING ARG)!
	01DA	A109	347	LD	R9,R0 !KST!
	01DC	030F	348	SUB	R15,#20
	01E0	1CF5	349	LEA	0R15,R1,#10 !SAVE REGS 1-10!
	01E4	A1A1	350	LD	R1,R10 !DPR NO PASSED IN R1!
	01EC	5F03	351	CALL	MM_ACTIVATE
			352	!R1:DER_NO,R2:PTR,R3:ENTRY_NO,	
			353	R4:SEGMENT_NO)!	
	01FA	5F00	354	! (RET:R0:SUCCESS_CODE,RR2:CLASS,R4:SIZE)!	
	01FE	942A	355	CALL	CCNFIRMMENT_CHECK ! (R0:SUCCESS_CODE)!
			356	LDL	RR10,RR2 !CLASS!
			357		
			358	!PAGE	

01F0 A14C	LD R12,R4	ISIZE!	359
01F2 1CF1	R1,RR15,#9	!RESTORE REGS 1-9!	360
01F6 A187	LD R7,R8	ISEG #!	361
01F8 0327	SUB R7,#NR OF KSEGS		362
01FC 1926	MULT RR6,#SIZEOF KST REC	!OFFSET TO REC!	363
0200 A17D	LD R13,R7		364
0202 619D	ADD R13,R8	!KST TO OFFSET!	365
0204 5DDA	LDL KST.CLASS(R13),RR10	!CLASS!	366
0206 6FLC	LD KST.SIZE(R13),R12	ISIZE!	367
0208 0406	CPB R10,#SUCCEEDED		368
0210 5E2E	IF EQ THEN		369
0214 93FD	PUSH RR15,R13		370
0216 5F00	CALL TC GET PROC CLASS		371
	!(RET:RR2:PROC CLASS)!		372
021A 97FD	POP R13,RR15		373
021C 54D4	LDL RR4,KST.CLASS(R13)		374
0220 97FD	PUSH RR15,R13		375
0222 91F2	PUSHL RR15,RR2		376
0224 91F4	PUSHL RR15,RR4		377
0226 5F00	CALL CLASS GE		378
	!(RR2:PROC CLASS,RR4:SEG CLASS,RR1:		379
	R1:CONDITION_CODE)!		380
022A 95F4	POPL RR4,RR15		381
022C 95F2	POPL RR2,RR15		382
022E 97FD	POP R13,RR15		383
0230 0F01	CP R1,#FALSE		384
0234 5E2E	IF EQ THEN	INO ACCESS POSSIBLE--DEACT.!	385
0238 1CF1	LDN R1,RR15,#10		386
023C A1A1	LD R1,R10	IDFR NCI	387
023E 76D2	LDA R2,KST.MM_HANDLE(R13)	!HPTF!	388
0242 5F00	CALL MM_DEACTIVATE	!RET:R0:S_CODE!	389
0246 5F00	CALL CONFINEMENT_CHECK	!R4:S_CODE!	390
024A 21F1	LD R1,RR15	ISEG #!	391
024C 2102	LD R2,#NULL	ACCESS	392
0250 2102	LD R0,#PRCC CLASS	NOT GF SIG CLASS	393
0254 5F08	ELSE		394
0256 93FD	PUSH RR15,R13		395

!	025A 5F02	0000*	397	CALL CLASS EQ 1(RR2:PROC_CLASS,
	025E 97FD		398	RR4:SEG CLASS,RET:R1:CONDITION_CODE))!
	0260 A110		399	PCP R13,CR15
	0262 1CF1	0108	400	LD R0,R1 !CONDITION_CODE!
	0266 0B00	0001	401	LDM R1,CR15,#9
	026A 5E0E	0282	402	CP R2,#TRUE
	026F 0P05	0000	403	IF EQ THEN
	0272 5E0E	027C	404	CP R5,#WRITE
	0276 CA00		405	IF EQ THEN
	027E 5E0E	027E	406	LD R2,RL2,#WRITE
	027C CA01		407	ELSE
			408	LD R2,RL2,#READ
			409	FI
	027F 5E0E	0284	410	FISE
	0282 CA01		411	LD R2,RL2,#READ
			412	FI
	0284 4C15	0009	413	LD R2,KST.IN_CORE(R13),#FALSE
	028E 0000			
	028A 6E0E	000E	414	LD R2,KST.M_SEG_NO(R13),RL6
	028F 6E0E	000F	415	LD R2,KST.ENTRY_NUMBER(R13),RL3
	0292 6EDA	0008	416	LDE R2,KST.ACCESS_MOLF(R13),RL2
	0296 2100	0002	417	LD R2,#SUCCEEDED !SUCCESS_CODE!
	029A 5F0E	02A6	418	FI
	029E 2101	FFFF	419	ELSE
	02A2 2102	0004	420	LD R1,#NULL_SEG
			421	LD R2,#NULL_ACCESS
	02AC 010F	0014	422	FI
	02AA 5F0E	02FA	423	ADD R15,#20
	02AE 2100	001B	424	ELSE
	02B2 2101	FFFF	425	LD R2,#NO_SEG_AVAIL
	02B6 2102	0014	426	LD R1,#NULL_SEG
			427	LD R2,#NULL_ACCESS
			428	FI
			429	FI
	02BA 9E0E		430	FI
	02BC		431	RFT
			432	END MAKE_KNOWN
			433	!PAGE

```

!
02FC
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
!PAGE

TERMINATE
PROCEDURE
!*****!
! CHECKS VALIDITY OF TERMINATE REQUEST !
! AND CALLS MM DEACTIVATE IF VALID !
!*****!
! REGISTER USE !
! PARAMETERS !
! R1:SEGMENT NO(INPUT) !
! R2:SUCCESS CODE(RETURNED) !
! LOCAL USE !
! R3:KST REC INDEX !
! R6:CONSTANT STORAGE !
! R13:~KST !
!*****!
ENTRY
LD R3,R1 ICOPY OF SEG #1
SUB R3,#NR_OF_KSEGS I CONVERT SEG# TO KST INDEX!
MULT RR2,#SIZECF KST_REC
PUSH @R15,R1
PUSH @R15,R3
LD R1,#KST_SEG_NO
CALL ITC_GET_SEG_PTR !(R1:KST_SEG_NO)!
!(RETURNS:R6:KST_SEG_PTR,!
LD R13,R2
POP R3,@R15
POP R1,@R15
ADD R13,R3 IADD OFFSET TO KST!
LD R6,#HULL_SEG
CPR R16,KST.M_SEG_NO(R13)
IF EQ THEN
LL R6,#SEGMENT NOT KNOWN
ELSE

```


! 033A

033A	A117		501
033C	0307	000A	502
0340	1906	0010	503
0344	93F1		504
0346	93F7		505
0348	2101	0002	506
034C	5F00	0000*	507
0350	A10D		508
0352	97F7		509
0354	97F1		510
0356	817D		511
0358	2106	FFFF	512
035C	4ADE	000E	513
0360	5F0F	036C	514
0364	2100	001C	515
0368	5E08	03AA	516
			517
			518
			519
			520
			521
			522
			523
			524
			525
			526
			527
			528
			529
			530
			531
			532
			533
			534
			535

```

SM_SWAP IN          PROCEDURE
!*****
! CHECKS VALIDITY OF SWAP IN REQUEST
! AND CALLS MM_SWAP IN IF VALID
!*****
! REGISTER USE
! PARAMETERS
! R1:SEGMENT NO(INPUT)
! R0:SUCCESS CODE(RETURNED)
! LOCAL USE
! R7:KST REC INDEX
! R3:ACCESS MODE
! R6:CONSTANT STORAGE
! R13:~KST
!*****
ENTRY
LD R7,R1      ICOPY OF SEG #1
SUB R7,#NR OF KSEGS  I CONVERT SEG# TO KST INDEX!
MULT R6,#SIZEOF KST REC  I OFFSET TO KST REC!
PUSH @R15,R1  I SAVE SEGMENT#!
PUSH @R15,R7
LD R1,#KST SEG_NO
CALL ITC GET SEG_PTR  I(R1:KST_SEG NO)!
LD R13,R4      I~KST!
POP R7,R15
POP R1,@R15  I RETRIEVE SEGMENT#!
ALD R13,R7  I ALD OFFSET TO KST EASF ADDR!
LD R6,#NULL_SEG
CPB R16,AST.M_SEG NO(R13)
IF EQ THEN
LD R2,#SEGMENT NOT KNOWN
ELSE

```

!	036C 2106	0001	LD R6,#TRUE	536
	0370 4ADE	0009	CPR RLC,KST.IN CORE(R13)	537
	0374 5E01	0380	IF EQ THEN	538
	037E 2102	0002	LD R0,#SUCCEEDED	539
	037C 5E08	03AA	ELSE	540
	0380 93FD		PUSH @R15,R13 !SAVE KST RFC ADDR!	541
	0382 5F00	0000*	CALL TC GET DBK NO !R1:(RET)DBK_NO!	542
	0386 97FD		POP R13,@R15	543
	038E 76D2	0000	LDA R2,KST.MM HANDLE(R13)	544
	0390 60DB	0008	LDE R13,KST.ACCESS MODE(R13)	545
	0390 93FD		PUSH @R15,R13 !SAVE SEG KST REC ADDR!	546
	0392 5F00	0000*	CALL MM_SWAP IN !R1:DPR_NO) !	547
			!R2:MM_HANDLE)!	548
			!R3:ACCESS_MODE)!	549
			!R0:(RET)SUCCESS_CODE)!	550
	0396 5F00	041A	CALL CONFINEMENT_CHECK !R0:SUCCESS_CODE)!	551
	039A 97FD		POP R13,@R15	552
	039C 0A08	0202	CPR R10,#SUCCEEDED	553
	03A0 5E0E	03AA	IF EQ THEN	554
	03A4 4CD5	0009	LDP KST.IN CORE(R13),#TRUE	555
	03A8 0101			556
			FI	557
			FI	558
	03AA 9E08		RET	559
	03AC		END SM_SWAP IN	560
			!PAGE	561

! 03AC

SM SWAP OUT

PROCEDURE

```

562 *****
563 *****
564 *****
565 ! CHECKS VALIDITY OF SWAP OUT REQUEST !
566 ! AND CALIS MM SWAP OUT IF VALID !
567 *****
568 ! REGISTER USE !
569 ! PARAMETERS !
570 ! R1:SEGMENT NO !
571 ! R0:SUCCESS CODE(RETURNED) !
572 ! LOCAL USFS !
573 ! R7:KST REC INDEX !
574 ! R6:CONSTANT STORAGE !
575 ! R13:~KST !
576 *****
577 *****

```

ENTRY

```

03AC A117      R7,R1 !COPY OF SEG #!
03AE 0307      R7,#NR OF KSEGS !CONVERT SEG# TO KST INDEX!
03P2 1006      MULT RR6,#SIZE OF KST REC !OFFSET TO KST_REC!
03R6 93F1      PUSH @R15,R1 !SAVE SEGMENT#!
03B8 93F7      PUSH @R15,R7
03PA 2101      LI R1,#KST SEG_NO
03BE 5F00      CALL ITC_GET_SEG_PTR ! (R1:KST_SEG_NO)!
03C2 A10D      LD R13,R0 !~KST!
03C4 97F7      POP R7,@R15
03C6 97F1      POP R1,@R15 !RETRIEVE SEGMENT#!
03C8 817D      ADD R13,R7 !ADD OFFSET TO KST BASE ADDR!
03CA 2106      LD R6,#NULL_SEG
03CE 4ADE      CPM R16,AST.M_SEG_NO(R13)
03D2 5E2E      IF EQ THEN
03DC 2101      LD R13,R6
03DA 5E2E      ELSE
03DE 2106      LD R6,#FALSE

```

1 PAGE

23F2	4ADE	0009	597	CPB	RL0,KST.IN CORE(R13)
03E6	5E0E	03F2	598	IF EQ THEN	
03FA	2104	00E2	599	LD	R0,#SUCCEEDED
03EE	5E06	0418	600	ELSE	
03F2	93FD		601	PUSH	GR15,R13 ISAVE KST REC ADDR!
03F4	5F04	0000*	602	CALL	TC GET DPR NO !R1:(RFT)DPR_NO!
03F8	97FD		603	POP	R13,R15
03FA	76D2	0000	604	LDA	R2,KST.MM_HANDLE(R13)
03FE	93FL		605	PUSH	GR15,R13 ISAVE SEG KST REC ADDR!
0400	5F00	0000*	606	CALL	MM_SWAP_OUT ! (R1:DPR_NO) !
			607		! (R2:MM_HANDLE) !
0404	5F00	041A	608	CALL	CONFINEMENT_CHECK ! (R0:SUCCESS_CODE) !
040E	97FD		609	POP	R13,GR15
040A	0A05	0002	610	CPE	RL0,#SUCCEEDED
040E	5E0E	0418	611	IF	EQ THEN
0412	4CD5	0009	612	LDR	KST.IN CORE(R13),#FALSE
0416	0000		613		
			614	FI	
			615	FI	
			616	FI	
			617	RET	
0418	9E08		618	END SM_SWAP_OUT	
041A			619		
			620	!PAGE	

APPENDIX C - DISTRIBUTED MEMORY MANAGER PLZ.SYS LISTINGS

DIST_MMGR MODULE

CONSTANT

```

CREATE_ENTRY_CODE      := 50
DELETE_ENTRY_CODE      := 51
ACTIVATE_SEG_CODE      := 52
DEACTIVATE_SEG_CODE    := 53
SWAP_IN_SEG_CODE       := 54
SWAP_OUT_SEG_CODE      := 55
NO_OF_PROCESSORS       := 1
MAX_NO_KST_ENTRIES     := ??
MAX_SEG_SIZE           := ??
MAX_DBR_NO             := 4
NR_OF_KSEGS            := ??
KST_SEG_NO             := ??

```

TYPE

```

H_ARRAY      ARRAY [3  WORD]

COM_MSG      ARRAY [16  BYTE]

CREATE_MSG    RECORD [CREATE_CODE  WORD
                      CE_MM_HANDLE H_ARRAY
                      CE_ENTRY_NO  SHORT_INTEGER
                      CE_FILLER    BYTE
                      CE_SIZE      WORD
                      CE_CLASS     LONG]

DELETE_MSG    RECORD [DELETE_CODE  WORD
                      DE_MM_HANDLE H_ARRAY
                      DE_ENTRY_NO  SHORT_INTEGER
                      DE_FILLER    ARRAY[7 BYTE]]

ACTIVATE_MSG  RECORD [ACTIVATE_CODE WORD
                      A_DBR_NO     SHORT_INTEGER
                      A_FILLER1     BYTE
                      A_MM_HANDLE   H_ARRAY
                      A_ENTRY_NO    SHORT_INTEGER
                      A_SEGMENT_NO  SHORT_INTEGER
                      A_FILLER2     LONG]

DEACTIVATE_MSG RECORD [DEACTIVATE_CODE WORD
                      D_DBR_NO     SHORT_INTEGER
                      D_FILLER1     BYTE
                      D_MM_HANDLE   H_ARRAY
                      D_FILLER2     ARRAY[3 WORD]]

```

```

SWAP_IN_MSG      RECORD [SWAP_IN_CODE  WORD
                        SI_MM_HANDLE  H_ARRAY
                        SI_DBR_NO    SHORT_INTEGER
                        SI_ACCESS_AUTH BYTE
                        SI_FILLER     ARRAY[3 WORD]]

SWAP_OUT_MSG     RECORD [SWAP_OUT_CODE WORD
                        SO_DBR_NO    SHORT_INTEGER
                        SO_FILLER1   BYTE
                        SO_MM_HANDLE  H_ARRAY
                        SO_FILLER2   ARRAY[3 WORD]]

R_SUC_CODE       RECORD [SUC_CODE      BYTE
                        SC_FILLER      ARRAY[15 BYTE]
                        ]

R_ACTIVATE_ARG   RECORD [R_SUC_CODE    BYTE
                        R_FILLER        BYTE
                        R_MM_HANDLE     H_ARRAY
                        R_CLASS         LONG
                        R_SIZE          WORD]

CE_PTR           ^CREATE_MSG

DE_PTR           ^DELETE_MSG

A_PTR            ^ACTIVATE_MSG

D_PTR            ^DEACTIVATE_MSG

SI_PTR           ^SWAP_IN_MSG

SO_PTR           ^SWAP_OUT_MSG

SC_PTR           ^R_SUC_CODE

ARG_PTR          ^R_ACTIVATE_ARG

KST_REC          RECORD [ MM_HANDLE     H_ARRAY
                        SIZE            WORD
                        ACCESS_MODE     BYTE
                        IN_CORE         BYTE
                        CLASS           LONG
                        M_SEG_NO        SHORT_INTEGER
                        ENTRY_NUMBER    SHORT_INTEGER ]

KST              ARRAY [ MAX_NO_KST_ENTRIES KST_REC ]

```

KSTPTR	^KST
ADDRESS	WORD
SEG_DESC_REG	RECORD [BASE_ADDR ADDRESS LIMIT BYTE ATTRIBUTE BYTE]
MMU	RECORD [SDR ARRAY [NO_SEG_DESC_REG SEG_DESC_REG] BLKS_USED WORD MAX_BIAS WORD]
MM_VP_ID	WORD
SEG_ARRAY	ARRAY [MAX_SEG_SIZE BYTE]
SEGPTR	^SEG_ARRAY

INTERNAL

MM_CPU_TABLE	ARRAY [NO_OF_PROCESSORS MM_VP_ID]
--------------	-----------------------------------

EXTERNAL

MMU_IMAGE	ARRAY [MAX_DBR_NO MMU]
G_AST_LOCK	BYTE
K_LOCK	PROCEDURE
K_UNLOCK	PROCEDURE
ITC_GET_CPU_NO	PROCEDURE
SIGNAL	PROCEDURE
WAIT	PROCEDURE

```

!*****
*
* MM_CREATE_ENTRY PROCEDURE. INVOZED BY SEGMENT
* MANAGER'S CREATE SEGMENT PROCEDURE. PERFORMS TYPE
* CONVERSION OF POINTERS, LOADS MESSAGE ARRAY FOR
* IPC, AND PERFORMS IPC WITH MEMCRY MANAGER PROCESS.
* RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_CREATE_ENTRY  PROCEDURE  (  HPTR          ^H_ARRAY
                             ENTRY_NO       SHORT_INTEGER
                             SIZE          WORD
                             CLASS        LONG )
                             RETURNS      ( SUCCESS_CODE  BYTE )

```

```

! **** NOTE:    REENTRANT PROCEIURE    **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL  CE_MSGPTR  CE_PTR
        COM_MSGBUF  COM_MSG
        COM_MSGPTR  ^COM_MSG
        SUC_CODE_PTR  SC_PTR

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERT TO OVERLAY ARGUMENT LIST ONTO MSG LIST !
  CE_MSGPTR := CE_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  CE_MSGPTR^.CREATE_CODE := CREATE_ENTRY_CODE
  CE_MSGPTR^.CE_MM_HANDLE[0] := HPTR^[2]
  CE_MSGPTR^.CE_MM_HANDLE[1] := HPTR^[1]
  CE_MSGPTR^.CE_MM_HANDLE[2] := EPTR^[2]
  CE_MSGPTR^.CE_ENTRY_NO := ENTRY_NO
  CE_MSGPTR^.CE_CLASS := CLASS
  CE_MSGPTR^.CE_SIZE := SIZE
  PERFORM_IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
  SUC_CODE_PTR := SC_PTR COM_MSGPTR
  SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
  RETURN
END MM_CREATE_ENTRY

```

```

!*****
*
* MM_DELETE_ENTRY PROCEDURE. INVOKED BY SEGMENT
* MANAGER'S DELETE_SEGMENT PROCEDURE. PERFORMS TYPE
* CONVERSION OF PCINTERS, LOADS MESSAGE ARRAY FOR
* IPC, AND PERFORMS IPC WITH MEMORY MANAGER PROCESS.
* RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_DELETE_ENTRY PROCEDURE ( HPTR      ^H_ARRAY
                           ENTRY_NO   SECRET_INTEGER)
                           RETURNS    ( SUCCESS_CODE BYTE )

```

```

! **** NOTE: REENTRANT PROCEDURE **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL  DE_MSGPTR  DE_PTR
        COM_MSGBUF COM_MSG
        COM_MSGPTR ^COM_MSG
        SUC_CODE_PTR SC_PTR

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERSION TO OVERLAY ARG LIST ONTO MSG ARRAY !
  DE_MSGPTR := LE_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  DE_MSGPTR^.DELETE_CODE := DELETE_ENTRY_CODE
  DE_MSGPTR^.DE_MM_HANDLE[0] := HPTR^[0]
  DE_MSGPTR^.DE_MM_HANDLE[1] := HPTR^[1]
  DE_MSGPTR^.DE_MM_HANDLE[2] := HPTR^[2]
  DE_MSGPTR^.DE_ENTRY_NO := ENTRY_NO
  PERFORM_IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO ARG LIST !
  SUC_CODE_PTR := SC_PTR COM_MSGPTR
  SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
  RETURN
END MM_DELETE_ENTRY

```

```

!*****
*
* MM_ACTIVATE PROCEDURE. INVOKED BY SEGMENT MANA-
* GER'S MAKE_ANCWN PROCEDURE. PERFORMS TYPE CONVERSION
* OF POINTERS, LOADS MESSAGE ARRAY FOR IPC, AND UP-
* DATES MM_HANDLE ENTRY IN KST AFTER PERFORMING THE
* IPC. RETURNS SUCCESS_CODE, CLASS, AND SIZE.
*
*****!

```

```

MM_ACTIVATE PROCEDURE ( DBR_NO          SHORT_INTEGER
                        HPTR            ^H_ARRAY
                        ENTRY_NO        SHORT_INTEGER
                        SEGMENT_NC      SHORT_INTEGER )
RETURNS ( SUCCESS_CODE BYTE
          CLASS          LONG
          SIZE           WORD )

```

```

! **** NOTE: REENTRANT PROCEDURE **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL  A_MSGPTR  A_PTR
        COM_MSGBUF  COM_MSG
        COM_MSGPTR ^COM_MSG
        RET_ARGPTR  ARGPTR
        KPTR        KSTPTR
        INDEX       SHORT_INTEGER

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY !
  A_MSGPTR := A_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  A_MSGPTR^.ACTIVATE_CODE := ACTIVATE_SEG_CODE
  A_MSGPTR^.A_DBR_NO := DBR_NC
  A_MSGPTR^.A_MM_HANDLE[2] := HPTR^[2]
  A_MSGPTR^.A_MM_HANDLE[1] := HPTR^[1]
  A_MSGPTR^.A_MM_HANDLE[2] := EPTR^[2]
  A_MSGPTR^.A_ENTRY_NO := ENTRY_NO
  A_MSGPTR^.A_SEGMENT_NO := SEGMENT_NO
  PERFORM IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
  RET_ARGPTR := ARG_PTR COM_MSGPTR
  SUCCESS_CODE := RET_ARGPTR^.R_SUC_CODE
  CLASS := RET_ARGPTR^.R_CLASS
  SIZE := RET_ARGPTR^.R_SIZE
  INDEX := SEGMENT_NC - NR_OF_KSEGS
  ! RETRIEVE HANDLE AND UPDATE KST !
  KPTR := KSTPTR MM_GET_KSEG_PTR ( SEG_NO )
  EPTR := #KPTR^[INDEX].MM_HANDLE
  HPTR^[0] := RET_ARGPTR^.R_MM_HANDLE[2]
  HPTR^[1] := RET_ARGPTR^.R_MM_HANDLE[1]
  HPTR^[2] := RET_ARGPTR^.R_MM_HANDLE[2]
  RETURN
END MM_ACTIVATE

```

```

!*****
*
*   MM_DEACTIVATE PROCEDURE. INVOKED BY SEG MGR'S
*   DEACTIVATE PROCEDURE. PERFORMS TYPE CONVERSION
*   OF POINTERS, LOADS MESSAGE ARRAY FOR IPC, AND PER-
*   FORMS IPC. RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_DEACTIVATE  PROCEDURE ( DBR_NO      SHORT_INTEGER
                          HPTR        ^H_ARRAY )
                      RETURNS ( SUCCESS_CODE BYTE )

```

```

! **** NOTE:   REENTRANT PROCEDURE   **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL  D_MSGPTR  D_PTR
        COM_MSGBUF COM_MSG
        COM_MSGPTR ^COM_MSG
        SUC_CODE_PTR SC_PTR

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY !
  D_MSGPTR := D_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  D_MSGPTR^.DEACTIVATE_CODE := DEACTIVATE_SEG_CODE
  D_MSGPTR^.D_DBR_NO := DBR_NO
  D_MSGPTR^.D_MM_HANDLE[2] := HPTR^[2]
  D_MSGPTR^.D_MM_HANDLE[1] := HPTR^[1]
  D_MSGPTR^.D_MM_HANDLE[2] := HPTR^[2]
  PERFORM_IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
  SUC_CODE_PTR := SC_PTR COM_MSGPTR
  SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
  RETURN
END MM_DEACTIVATE

```

```

!*****
*
* MM_SWAP_IN PROCEDURE. INVOKED BY SEGMENT MANAGER'S
* SM_SWAP_IN PROCEDURE. PERFORMS TYPE CONVERSION OF
* POINTERS, LOADS MESSAGE ARRAY FOR IPC, AND PERFORMS
* IPC. RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_SWAP_IN  PROCEDURE ( DBR_NO          SHORT_INTEGER
                       HPTR             ^E_ARRAY
                       ACCESS_MODE      BYTE )
                RETURNS ( SUCCESS_CODE  BYTE )

```

```

! **** NOTE:      REENTRANT PROCEDURE      **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !
LOCAL  SI_MSGPTR  SI_PTR
       COM_MSGBUF  COM_MSG
       COM_MSGPTR  ^COM_MSG
       SUC_CODE_PTR SC_PTR

```

```

ENTRY
  COM_MSGPTR := #COM_MSGBUF
  ! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY !
  SI_MSGPTR := SI_PTR COM_MSGPTR
  ! LOAD ARG LIST ONTO MSG ARRAY !
  SI_MSGPTR^.SWAP_IN_CODE := SWAP_IN_SEG_CODE
  SI_MSGPTR^.SI_MM_HANDLE[0] := HPTR^[0]
  SI_MSGPTR^.SI_MM_HANDLE[1] := HPTR^[1]
  SI_MSGPTR^.SI_MM_HANDLE[2] := HPTR^[2]
  SI_MSGPTR^.SI_DBR_NO := DBR_NO
  SI_MSGPTR^.SI_ACCESS_AUTH := ACCESS_MODE
  PERFORM IPC (COM_MSGPTR)
  ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
  SUC_CODE_PTR := SC_PTR COM_MSGPTR
  SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
  RETURN
END MM_SWAP_IN

```

```

!*****
*
*   MM_SWAP_OUT PROCEDURE. INVOKED BY SEGMENT MANAGER'S
*   SM_SWAP_OUT PROCEDURE. PERFORMS TYPE CONVERSION OF
*   PCINTERS, LOADS MESSAGE ARRAY FOR IPC, AND PERFORMS
*   IPC. RETURNS SUCCESS_CODE.
*
*****!

```

```

MM_SWAP_OUT   PROCEDURE ( HPTR           ^H_ARRAY
                        ACCESS_MODE  BYTE )
                        RETURNS      ( SUCCESS_CODE  BYTE )

```

```

! **** NOTE:   REENTRANT   PROCEDURE   **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL   SO_MSGPTR   SO_PTR
        COM_MSGBUF   COM_MSG
        COM_MSGPTR   ^COM_MSG
        SUC_CODE_PTR SC_PTR

```

```

ENTRY

```

```

    COM_MSGPTR := #COM_MSGBUF
    ! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY !
    SO_MSGPTR := SO_PTR COM_MSGPTR
    ! LOAD ARG LIST ONTO MSG ARRAY !
    SO_MSGPTR^.SWAP_OUT_CODE := SWAP_OUT_SEG_CODE
    SO_MSGPTR^.SO_MM_HANDLE[0] := EPTR^[0]
    SC_MSGPTR^.SC_MM_HANDLE[1] := LPTR^[1]
    SO_MSGPTR^.SO_MM_HANDLE[2] := HPTR^[2]
    SO_MSGPTR^.SO_DBR_NO := DBR_NO
    PERFORM IPC (COM_MSGPTR)
    ! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !
    SUC_CODE_PTR := SC_PTR COM_MSGPTR
    SUCCESS_CODE := SUC_CODE_PTR^.SUC_CODE
    RETURN
END MM_SWAP_OUT

```

```

!*****
*
*   MM GET_DBR_VALUE. SERVICE ROUTINE THAT RETRIEVES
*   THE DBR "VALUE" (IE ADDRESS) BASED UPON A DBR_NO.
*
*****!

```

```

MM_GET_DBR_VALUE    PROCEDURE (DBR_NO    SHORT_INTEGER)
                    RETURNS      (DBR_VALUE ADDRESS)

```

```

! **** NOTE:    REENTRANT PROCEDURE    **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

ENTRY
    DBR_VALUE := #MMU_IMAGE [DBR_NO]
    RETURN
END MM_GET_DBR_VALUE

```

```

!*****
*
*   PERFORM_IPC PROCEDURE. INVOKED BY DISTRIBUTED
*   MEMORY MANAGER PROCEDURES. DETERMINES CURRENT
*   MEMORY MANAGER'S VP_ID, LOCKS G_AST, PERFORMS
*   IPC VIA SIGNAL AND WAIT PRIMITIVES, AND UNLOCKS
*   THE G_AST.
*
*****!

```

```

PERFORM_IPC    PROCEDURE (COM_MSGPTR    ^COM_MSG)

```

```

! **** NOTE:    REENTRANT PROCEDURE    **** !
! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

```

```

LOCAL    MMGR_VP_ID

ENTRY
    ! DETERMINE MMGR VP_ID !
    CPU_NO := ITC_GET_CPU_NO
    MMGR_VP_ID := MM_CPU_TABLE [CPU_NO]
    K_LOCK (#G_AST_LOCK)
    ! IPC WITH MEMORY MANAGER PROCESS !
    SIGNAL ( COM_MSGPTR, MMGR_VP_ID )
    WAIT ( COM_MSGPTR )
    K_UNLOCK (#G_AST_LOCK )
    RETURN
END PERFORM_IPC

END DIST_MMGR

```

APPENDIX D - DISTRIBUTED MEMORY MANAGER PLZ/ASM LISTINGS

DIST_MM MODULE

CONSTANT

```

CREATE_ENTRY_CODE      := 50
DELETE_ENTRY_CODE      := 51
ACTIVATE_SEG_CODE      := 52
DEACTIVATE_SEG_CODE    := 53
SWAP_IN_SEG_CODE       := 54
SWAP_OUT_SEG_CODE      := 55
NO_OF_PROCESSORS       := 1
MAX_NO_KST_ENTRIES     := 54
MAX_SEG_SIZE           := 128
MAX_DBR_NO             := 4
KST_SEG_NO             := 2
NR_OF_KSEGS            := 10

```

TYPE

H_ARRAY ARRAY [3 WORD]

COM_MSG ARRAY [16 BYTE]

KST_REC RECORD [MM_HANDLE F_ARRAY

 SIZE WORD

 ACCESS_MODE BYTE

 IN_CORE BYTE

 CLASS LONG

 M_SEG_NO SHORT_INTEGER

 ENTRY_NUMBER SHORT_INTEGER]

 ADDRESS WORD

36 !PAGE

```

37 SEG_DESC_REG RECORD [BASE_ADDR ADDRESS
38 LIMIT BYTE
39 ATTRIBUTE BYTE]
40
41 MMU RECORD [SDR_ARRAY [64
42 SEG_DESC_REG]]
43 INOTE: NEXT TWO COMPONENTS
44 LEFT OUT FOR CONVENIENCE SINCE
45 ARE NOT USED FOR SEGMENT MGR
46 DEMO!
47
48
49
50
51 MM_VP_ID WORD
52
53 SEG_ARRAY ARRAY [MAX_SEG_SIZE BYTE]
54 INTERNAL
55
56 $SECTION D_MM_DATA
57 MM_CPU_TABLE_ARRAY [NO_OF_PROCESSORS MM_VP_ID] :=[L]
58
59 $SECTION MSG_FRAME_DCL
60 INOTE: THESE RECORDS ARE "OVERLAYS" OR "FRAMES" USED
61 TO DEFINE MESSAGE FORMATS. NO MEMORY IS ALLOCATED FOR
62 THEM!
63 $ABS 2
64 CREATE_MSG RECORD [CREATE_CODE WORD
65 CE_MM_HANDLE E_ARRAY
66 CE_ENTRY_NO SHORT_INTEGER
67 CE_FILLER BYTE
68 CE_SIZE WORD
69 CE_CLASS LONG]
70
71 !PAGE

```

0000 0000

0000

72	\$ABS 0				
73	DELETE_MSG	RECORD	[DELETE_CODE	WORD	
74			DE_MM_HANDLE	E_ARRAY	
75			DE_ENTRY_NO	SHORT_INTEGER	
76			DE_FILLER	ARRAY[7 BYTE]]	
77					
78	\$ABS 0				
79	ACTIVATE_MSG	RECORD	[ACTIVATE_CODE	WORD	
80			A_DBR_NO	SHORT_INTEGER	
81			A_FILLER1	PYTE	
82			A_MM_HANDLE	H_ARRAY	
83			A_ENTRY_NO	SHORT_INTEGER	
84			A_SEGMENT_NO	SHORT_INTEGER	
85			A_FILLER2	LONG]	
86					
87	\$ABS 0				
88	DEACTIVATE_MSG	RECORD	[DEACTIVATE_CODE	WORD	
89			D_DBR_NO	SHORT_INTEGER	
90			D_FILLER1	BYTE	
91			D_MM_HANDLE	H_ARRAY	
92			D_FILLER2	ARRAY[3 WORD]]	
93					
94	\$ABS 0				
95	SWAP_IN_MSG	RECORD	[SWAP_IN_CODE	WORD	
96			SI_MM_HANDLE	E_ARRAY	
97			SI_DBR_NO	SHORT_INTEGER	
98			SI_ACCESS_AUTH	BYTE	
99			SI_FILLER	ARRAY[3 WORD]]	
100					
101	\$ABS 0				
102	SWAP_OUT_MSG	RECORD	[SWAP_OUT_CODE	WORD	
103			SO_DBR_NO	SHORT_INTEGER	
104			SO_FILLER1	PYTE	
105			SO_MM_HANDLE	H_ARRAY	
106			SO_FILLER2	ARRAY[3 WORD]]	

IPAGE

1	127	\$ABS 0			
0000	128	RET_SUC_CODE	RECORD [SUC_CODE	BYTE	
	129		SC_FILLER	ARRAY[15 BYTE]	
	130		]		
	131				
	132	\$ABS 0			
0000	133	R_ACTIVATE_ARG	RECORD [R_SUC_CODE	BYTE	
	134		R_FILLER	BYTE	
	135		R_MM_HANDLE	H_ARRAY	
	136		R_CLASS	LONG	
	137		R_SIZE	WORD	
	138		R_FILLER1	WORD]	
	139				
	140	\$ABS 0			
0000	141	KST	ARRAY [MAX_NO_KST_ENTRIES KST_REC]		
	142				
	143	EXTERNAL			
	144				
	145	MMU_IMAGE	ARRAY [MAX_DER_NO MMU]		
	146				
	147	G_AST_LOCK	WORD		
	148				
	149	K_LOCK	PROCEDURE		
	150				
	151	K_UNLOCK	PROCEDURE		
	152				
	153	ITC_GET_CPU_NO	PROCEDURE		
	154				
	155	SIGNAL	PROCEDURE		
	156				
	157	WAIT	PROCEDURE		
	158				
	159	ITC_GET_SEG_PTR	PROCEDURE		
	160				
	161	!PAGE			

```

142 GLOBAL
143 $SECTION D_MM_PLOC
144 MM_CREATE_ENTRY
145 !*****PROCEDURE*****!
146 ! INTERFACE BETWEEN SEG MGR !
147 ! (CREATE_SEG PROCEDURE) AND !
148 ! MMGR PROCESS (CREATE_ENTRY !
149 ! PROCEDURE). ARRANGES AND !
150 ! PERFORMS IPC. !
151 !*****PROCEDURE*****!
152 ! REGISTER USE: !
153 ! PARAMETERS !
154 ! R2:SUCCESS CODE (RET) !
155 ! R1:HPTR (INPUT) !
156 ! R2:ENTRY NO (INPUT) !
157 ! R3:SIZE (INPUT) !
158 ! R4:CLASS (INPUT) !
159 ! LOCAL USE !
160 ! R6:MM_HANDLE_ARRAY ENTRY !
161 ! R8:COM_MSGBUF !
162 ! R13:COM_MSGBUF !
163 !*****PROCEDURE*****!
164 ENTRY
165 SUP R15,#SIZEOF COM_MSG IUSE STACK FOR MESSAGE!
166 LD R13,R15 ! COM_MSGBUF !
167
168 ! FILL COM_MSGBUF (LOAD MESSAGE). CREATE_MSG FRAME
169 ! IS EASFD AT ADDRESS ZERO. IT IS OVERLAID ONTO
170 ! COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
171 ! ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
172
173 LD CREATE_MSG.CREATE_CODE(P13),#CREATE_ENTRY_CODE
174
175 LD R6,R1(#0) ! INDEX TO MM_HANDLE ENTRY!
176 LD CREATE_MSG.CE_MM_HANDLE[0](R13),R6
177 LD R6,R1(#2)
177 !PAGE

```

!	0018	6FD6	0004	LD	CREATE_MSG.CE_MM_HANDLE[1](R13),R6
	001C	3116	0004	LD	R6,R1(#4)
	0020	6FD6	0006	LD	CREATE_MSG.CE_MM_HANDLE[2](R13),R6
	0024	6FD2	0008	LD	CREATE_MSG.CE_ENTRY_NO(R13),R2
	002E	5DD4	000C	LDL	CREATE_MSG.CE_CLASS(R13),RR4
	002C	6FD3	000A	LD	CREATE_MSG.CE_SIZE(R13),R3
	0030	A1DE		LD	R6,R13
	0032	5F00	01A8	CALL	PERFORM_IPC IR6: ^COM_MSGBUF!
	003C	60DE		IRETRIEVE	SUCCESS CODE FROM RETURNED MESSAGE!
	003A	010F	0010	LDB	RL6,RET_SUC_CODE.SUC_CODE(R13)
	003E	9E08		ADD	R15,#SIZEOF_COM_MSG !RESTORE STACK STATE!
	0040			RET	
				END MM_CREATE_ENTRY	
				191 !PAGE	

```

!
0040
192 MM_DELETE ENTRY
193 !*****PROCEDURE*****!
194 ! INTERFACE BETWEEN SEG MGR !
195 ! (DELETE_SEG PROCEDURE) AND !
196 ! MMGR (DELETE_ENTRY PROCEDURE).!
197 ! ARRANGES AND PERFORMS IPC. !
198 !*****!
199 ! REGISTER USE: !
200 ! PARAMETERS !
201 ! R0:SUCCESS CODE(RET) !
202 ! R1:HPTR(INPUT) !
203 ! R2:ENTRY_NO(INPUT, !
204 ! LOCAL USE !
205 ! R6:MM_HANDLE ARRAY ENTRY !
206 ! R8:~COM_MSGBUF !
207 ! R13:~COM_MSGBUF !
208 !*****!
209 ENTRY
210 SUB R15,#SIZEOF COM_MSG !USE STACK FOR MESSAGE!
211 LD R13,R15 ! COM_MSGBUF !
212
213 !FILL COM_MSGBUF (LOAD MESSAGE). DELETE MSG FRAME
214 IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
215 COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.e. ADD-
216 ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
217
218 LD DELETE_MSG.DELETE_CODE(R13),#DELETE_ENTRY_CODE
219
220 LD R6,R1(#0) !INDEX TO MM_HANDLE ENTRY!
221 LD DELETE_MSG.DE_MM_HANDLE[R6](R13),R6
222 LD R6,R1(#2)
223 LD DELETE_MSG.DE_MM_HANDLE[1](R13),R6
224 LD R6,R1(#4)
225 LD DELETE_MSG.DE_MM_HANDLE[2](R13),R6
226 LD DELETE_MSG.DE_ENTRY_NO(R13),R2
227 LD R6,R13
228 !PAGE

```

!	00CA 5F00	01A3	228	CALL PERFORM IPC	IR8: ^COM MSGBUF!
	006E 60D8	0000	229	IRETRIEVE SUCCESS	CODE FROM RETURNED MESSAGE!
	0072 010F	0010	230	LDB R10,RET	SUC_CODE(SUC_CODE(R13))
	0076 9F08		231	ADD R15,#SIZEOF	COM MSG
	0078		232	RET	IRESTORE STACA STATE!
			233	END MM_DELETE	ENTRY
			234	IPAGE	

! 007E

```

235 NM_ACTIVATE          PROCEDURE
236 !*****
237 ! INTERFACE BETWEEN SEG MGR
238 ! (MAKE_ANCWN PROCEDURE) AND
239 ! MMGR (ACTIVATE PROCEDURE).
240 ! ARRANGES AND PERFORMS IPC.
241 !*****
242 ! REGISTER USE:
243 ! PARAMETERS
244 ! R0:SUCCESS CODE(RET)
245 ! R1:DER NO(INPUT)
246 ! R2:HPTR(INPUT)
247 ! R3:ENTRY_NC
248 ! R4:SEGMENT_NO
249 ! LOCAL USE
250 ! R6:COM_MSGBUF
251 ! R13:COM_MSGBUF
252 !*****
253 !*****
254 !*****
255 ENTRY SUB R15,#SIZEOF COM_MSG IUSE STACK FOR MESSAGE!
256 LD R13,R15 ! COM_MSGBUF !
257
258 !FILL COM_MSGBUF (LOAD MESSAGE). ACTIVATE_MSG_FRAME
259 IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
260 COM_MSGBUF FRAME IF INDEXING EACH ENTRY (I.E. ADD-
261 ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF
262 LD ACTIVATE_MSG.ACTIVATE_CODE(R13),
263
264 #ACTIVATE_SEG_CODE
265 LDP ACTIVATE_MSG.A DER NO(R13),RL1
266 LD R0,R2(#0)
267 LD ACTIVATE_MSG.A MM_HANDLE[0](R13),R0
268 LD R6,R2(#2)
269 LD ACTIVATE_MSG.A MM_HANDLE[1](R13),R6
270 LD R0,R2(#4)
271
272 !PAGE

```

2078 030F 0010
007C A1FD

0071 4DD5 0000
0052 0334

00E4 61D9 00C2
0258 3126 0000
028C 6FD6 0034
0490 3126 0002
0294 6FD6 0026
0098 3126 0034

!	229C	6FD6	0208	LD	ACTIVATE MSG.A_MM_HANDLE[2](R13),R6
	22A2	6EDB	020A	LDR	ACTIVATE MSG.A_ENTRY_NO(R13),R13
	22A4	6EDC	020E	LDI	ACTIVATE MSG.A_SFGMENT_NO(R13),R14
	22A8	A1D8		LD	R6,R13
	22AA	93F4		PUSH	QR15,R4 ISAVE COPY SEG #!
	22AC	5F00	01AB	CALL	PERFCHM_IPC ! (R2:COM_MSGPUF!
	22B3	2101	2002	!	UPDATE KST_MM_HANDLE ENTRY!
	22B4	5F00	0300*	LD	R1,#KST_SEG_NO
	24FE	A10C		CALL	ITC_GET_SEG_PTR ! (R1:KST_SEG_NO,RET:R0:~KST)!
	22BA	97F4		LD	R12,R0 !~KST!
	22BC	A145		PCP	R4,QR15 !SFG #!
	24FE	0305	000A	LD	R5,R4 !MOVE SEG # TO ALLOW MULT!
	2002	1904	0210	SUB	R5,#NR OF KSEGS !CONVERT TO INDEX!
	2006	815C		MULT	RR4,#SIZEOF KST_REC !OFFSET IN KST TO SFG REC!
	24CE	61DC	0002	ADD	R12,R5 !ADD OFFSET TO ~KST!
	20CC	6FC6	0200	LD	R6,R_ACTIVATE_ARG.R_MM_HANDLE[0](R13)
	24D0	61DC	0004	LD	KST_MM_HANDLE[0](R12),R6
	22D4	6FC6	0202	LD	R6,R_ACTIVATE_ARG.R_MM_HANDLE[1](R13)
	22D8	61DC	0206	LD	KST_MM_HANDLE[1](R12),R6
	24DC	6FC6	0004	LD	R6,R_ACTIVATE_ARG.R_MM_HANDLE[2](R13)
				LD	KST_MM_HANDLE[2](R12),R6
				!	RETRIEVE CTHER RETURN ARGUMENTS!
	00E0	60D8	0000	LDR	RL0,R_ACTIVATE_ARG.R_SUC_CODE(R13)
	24F4	54D2	0008	LDL	RR2,R_ACTIVATE_ARG.R_CLASS(R13)
	00E8	61D4	020C	LD	R4,R_ACTIVATE_ARG.R_SIZE(P13)
	22EC	210F	0210	ADD	R15,#SIZEOF COM_MSG !RESTORE STACK STATE!
	22F2	9F2E		RET	
	22F2			END MM_ACTIVATE	
				300	IPAGE

! 20F2

```

301 MM_DEACTIVATE ***** PROCEDURE
302 ! ***** !
303 ! INTERFACE BETWEEN SEG MGR !
304 ! (TERMINATE PROCEDURE) AND !
305 ! MMGR (DEACTIVATE PROCEDURE). !
306 ! ARRANGES AND PERFORMS IPC. !
307 ! ***** !
308 ! REGISTER USE: !
309 ! PARAMETERS !
310 ! R0: SUCCESS CODE (RET) !
311 ! R1: DPR NO (INPUT) !
312 ! R2: EPR (INPUT) !
313 ! LOCAL USE !
314 ! R6: MM_HANDLE ARRAY ENTRY !
315 ! R8: COM_MSGBUF !
316 ! R13: COM_MSGBUF !
317 ! ***** !
318
319 ENTRY
320 SUR R15, #SIZEOF COM_MSG !USE STACK FOR MESSAGE!
321 LD R13, R15 ! COM_MSGBUF !
322
323 ! FILL COM_MSGBUF (LOAD MESSAGE). DEACTIVATE_MSG FRAME
324 ! IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
325 ! COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
326 ! ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
327
328 LD DEACTIVATE_MSG.DEACTIVATE_CCDI(R13),
329
329 #DEACTIVATE_SEG_CODE
330 LIB DEACTIVATE_MSG.D_LPR NO(R13), R11
331 LD R6, R2(#2) ! INDEX TO MM_HANDLE ENTRY!
332 LD DEACTIVATE_MSG.D_MM_HANDLE[R](R13), R6
333 LD R6, R2(#2)
334 LD DEACTIVATE_MSG.D_MM_HANDLE[1](R13), R6
335 LD R6, R2(#4)
336 ! PAGE

```

00F2 030F 0010
00F6 A1FE

00FE 4DD5 0002
00FC 003E

00FE 6ED9 0002
0102 3126 0000
0106 6FDE 0004
010A 3126 0002
010E 6FDE 0006
0112 3126 0004

```

3337 LD      DEACTIVATE MSG.D_MM_HANDLE[2](R13),R6
3338 LD      R6,R13
3339 CALL   PERFORM_IPC !R6: ^COM_MSGBUF!
3340
3341 !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!!
3342
3343 LDR      R10,RET_SUC_CODE,SUC_CODE(R13)
3344 ADD      R15,#SIZEOF_COM_MSG !RESTORE STACK ST
3345 RET
3346
3347 FND MM_LFACIVATE
3348 !PAGE

```

! 012A

```

346 MM_SWAP IN PPOCEDURE
349 !*****
350 ! INTERFACE BETWEEN SEG MGR (SM)
351 ! SWAP IN PROCEDURE) AND MMGR
352 ! (SWAP IN PROCEDURE). ARRANGES
353 ! AND PERFORMS IPC.
354 !*****
355 ! REGISTER USE:
356 ! PARAMETERS
357 ! R0:SUCCESS CODE(RET)
358 ! R1:DDR NO(INPUT)
359 ! R2:HPTA(INPUT)
360 ! R3:ACCESS_MODE(INPUT)
361 ! LOCAL USE
362 ! R6:MM_HANDLE_ARRAY ENTRY
363 ! R8:COM_MSGBUF
364 ! R13:COM_MSGBUF
365 !*****
366 ENTRY
367 SUB R15,#SIZEOF COM_MSG IUSE STACK FOR MESSAGE!
368 LD R13,R15 I COM_MSGBUF
369
370 ! FILL COM_MSGBUF (LOAD MESSAGE). SWAP IN MSG FRAME
371 ! IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
372 ! COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
373 ! ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF
374
375 LD SWAP IN MSG_SWAP IN COLI(R13),#SWAP IN SIG_COLI
376
377 LD R6,R2(#0) ! INDEX TO MM_HANDLE ENTRY
378 LD SWAP IN MSG.SI MM_HANDLE[0](R13),R6
379 LD R6,R2(#2)
380 LD SWAP IN MSG.SI MM_HANDLE[1](R13),R6
381 LD R6,R2(#4)
382 LD SWAP IN MSG.SI MM_HANDLE[2](R13),R6
383 LD SWAP IN MSG.SI DDR_NO(R13),R1
384
385 ! PAGE

```

!	0152	6ED3	0009	LDR	SWAP IN MSG.SI ACCESS AUTH(R13),R13
	0156	A1DE		LD	R8,R13
	0158	5F00	01A8	CALL	PERFORM_IPC IRE: ^COM MSGBUF!
	015C	62DE	0000		!RETRIEVE SUCCESS CODE FROM RETURNED MESSAGE!
	0160	010F	0010	LDE	R10,RET_SUC_CODE.SUC_CODE(R13)
	0164	9E08		ADD	R15,#SIZEOF_COM_MSG !RESTORE STACK STATE!
0166				RET	
					END MM SWAP IN
					392 !PAGE

! 2166

```

393 MM_SWAP_OUT PROCEDURE
394 !*****!
395 ! INTERFACE BETWEEN SEG MGR (SM_!
396 ! SWAP_OUT PROCEDURE) AND MMGR !
397 ! (SWAP_OUT PROCEDURE). ARRANGES!
398 ! AND PERFORMS IPC. !
399 !*****!
400 ! REGISTER USE: !
401 ! PARAMETERS !
402 ! R0:SUCCESS CODE(RET) !
403 ! R1:DBR_NO(INPUT) !
404 ! R2:HPTR(INPUT) !
405 ! LOCAL USE !
406 ! R6:MM_HANDLE_ARRAY ENTRY !
407 ! R2:COM_MSGBUF !
408 ! R13:COM_MSGBUF !
409 !*****!
410 ENTRY
411 SUB R15,#SIZEOF COM_MSG IUSE STACK FOR MESSAGE!
412 LD R13,R15 ! COM_MSGBUF !
413
414 ! FILL COM_MSGBUF (LOAD MESSAGE). SWAP_OUT MSG FRAME
415 ! IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
416 ! COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
417 ! ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
418
419 LD SWAP_OUT MSG_SWAP_OUT CODE(R13),
420
421 LD R6,R2(#4) ! INDEX TO MM_HANDLE ENTRY!
422 LD SWAP_OUT MSG_SO MM_HANDLE[0](R13),R6
423 LD R6,R2(#2)
424 LD SWAP_OUT MSG_SO MM_HANDLE[1](R13),R6
425 LD R6,R2(#4)
426 LD SWAP_OUT MSG_SO MM_HANDLE[2](R13),R6
427 LD SWAP_OUT MSG_SO DBR_NO(R13),R11
428 !PAGE

```

!	018E A1DE		LD R6,R13
	0190 5F00	01A8	CALL PERFORM_IPC !RE: ~COM_MSGBUF!
	0194 60DE	0000	!RETRIEVE SUCCESS CODE FROM RETURNED MESSAGE!
	0198 010F	0010	LDF R14,RET_SUC_CODE.SUC_CODE(R13)
	019C 9EF8		ADD R15,#SIZEOF_COM_MSG !RESTORE STACK STAFF!
	019F		RET
			END MM_SWAP_OUT
			436 !PAGE
			429
			430
			431
			432
			433
			434
			435

!
019E

```
437 MM GET_DBR_VALUE PROCEDURE
438 !*****!
439 ! RETRIEVES DBR VALUE (ADDRESS !
440 ! OF MMU REC POINTED TO BY DBR !
441 ! NO) !
442 !*****!
443 ! REGISTER USE: !
444 ! R1:DBR NO (INPUT) !
445 ! R2:DBR VALUE (RET) !
446 !*****!
447 !*****!
448 ENTRY
449 MULT RR0,#SIZEOF_MMU
450 LDA R2,MMU_IMAGE(R1)
451 RET
452 END MM_GET_DBR_VALUE
453
454 !PAGE
```

019E 1900 0100
01A2 7612 0000*
01A6 9E08
01A8

! 01A8

```

455      PERFORM IPC
456      !*****PROCEDURE*****!
457      ! SERVICE ROUTINE TO ARRANGE AND !
458      ! PERFORM IPC WITH THE MEM MGR PROC !
459      !*****!
460      ! REGISTER USE: !
461      ! PARAMETERS !
462      ! R6: ^COM_MSG(INPUT) !
463      ! LOCAL USE !
464      ! R1,R2: WORK REGS !
465      ! R4: ^G_AST_LOCK !
466      ! R13: ^COM_MSGBUF !
467      !*****!
468
469
470      ENTRY
471      PUSH GR15,R13 !^COM_MSGBUF!
472      CALL ITC_GET_CPU_NO !RIT-R1:CPU_NO!
473      LD R2,R1
474      LD R1,MM_CPU_TABLE(R2) IMM VP ID!
475      LDA R4,G_AST_LOCK
476      CALL K_LOCK
477      CALL SIGNAL IR1:MM_VP_ID,R6:^COM_MSGBUF!
478      POP R13,R15 !^COM_MSGBUF!
479      LD R8,R13
480      PUSH GR15,R13
481      CALL WAIT IR8:^COM_MSGBUF!
482      LDA R4,G_AST_LOCK
483      CALL K_UNLOCK
484      POP R13,R15
485      RET
486      END PERFORM_IPC
487
488      END DIST MM
489      !PAGE

```

```

01A8 93FD
01A9 5F00 0000*
01AA A112 0000*
01AB 6121 0000*
01AC 7604 0000*
01AD 5F00 0000*
01AE 5F00 0000*
01AF 97FD
01B0 A1D8
01B1 93FD
01B2 5F00 0000*
01B3 7604 0000*
01B4 5F00 0000*
01B5 97FD
01B6 9F00
01B7

```

APPENDIX E - NON-DISCRETIONARY SECURITY PL2/SYS LISTINGS

NDS MODULE

CONSTANT

```
TRUE      := 1
FALSE     := 0
```

TYPE

```
ACCESS_CLASS    RECORD [ LEVEL INTEGER
                        CAT   INTEGER ]
CPTR             ^ACCESS_CLASS
```

INTERNAL

```
CLASS1_PTR      CPTR
CLASS2_PTR      CPTR
CATS_REL        INTEGER
```

GLOBAL

```
!*****
*
* CLASS_EQ PROCEDURE. INVOKED BY VARIOUS KERNEL
* PROCEDURES. COMPARES TWO CLASSIFICATIONS (LABELS)
* AND DETERMINES IF THEIR RELATIONSHIP IS EQUAL.
* RETURNS A TRUE/FALSE CONDITION_CODE.
*
*****!
```

```
CLASS_EQ PROCEDURE ( CLASS1      LONG
                     CLASS2      LONG )
                     RETURNS ( CONDITION_CODE BYTE )
```

ENTRY

```
IF CLASS1 = CLASS2 THEN
    CONDITION_CODE := TRUE
ELSE
    CONDITION_CODE := FALSE
FI
RETURN
END CLASS_EQ
```

```

!*****
*
* CLASS_GE PROCEDURE. CALLED BY VARIOUS KERNEL
* PROCEDURES. COMPARES TWO CLASSIFICATIONS (CLASS1
* AND CLASS2) AND DETERMINES IF LABEL1 IS GREATER
* THAN OR EQUAL TO CLASS2. RETURNS TRUE/FALSE CONDI-
* TION CODE.
*
*****!

```

```

CLASS_GE PROCEDURE ( CLASS1      LONG
                   CLASS2      LONG )
                   RETURNS ( CONDITION_CODE BYTE )

```

```

ENTRY
! TYPE CONVERSION TO ALLOW OPERATIONS ON THE 16 MOST !
! AND 16 LEAST SIGNIFICANT BITS OF EACH CLASS. THE !
! 16 MSBITS ARE THE CLASS LEVEL AND THE 16 LSBITS !
! ARE THE CLASS CATEGORY (CAT). !

CLASS1_PTR := CPTR #CLASS1
CLASS2_PTR := CPTR #CLASS2
! COMPARE CATEGORIES (SET COMPARISON); SEE IF CAT2 !
! IS A SUBSET OF CAT1. !
CATS_REL := CLASS1_PTR^.CAT OR CLASS2_PTR^.CAT
IF CATS_REL = CLASS1_PTR^.CAT ! THEN CAT2 IS SUBSET !
ANDIF CLASS1_PTR^.LEVEL >= CLASS2_PTR^.LEVEL THEN
! LEVEL COMPARISON IS SIMPLE NUMERICAL COMPARISON !
CONDITION_CODE := TRUE
ELSE
CONDITION_CODE := FALSE
FI
RETURN
END CLASS_GE
END NDS

```


Address	Hex	Assembly	Comments
0014	91F2	ENTRY	
0016	A1FD	PUSHL R15,RR2 !PUSH CLASS1 ON STACK--REFER BY ADDR!	
0018	91F4	LD R13,R15 ! CLASS1 ADDR !	
001A	A1FF	PUSHL R15,RR4	
001C	31F7	LD R14,R15 ! CLASS2 ADDR !	
0020	45D7	LD R7,R14(#ACCESS_CLASS.CAT)	! CAT2 IN R7 !
0024	4FD7	OR R7,ACCESS_CLASS.CAT(R13)	!CAT1 OR CAT2, R7!
0028	5E3E	CP R7,ACCESS_CLASS.CAT(R13)	!CAT1=(CAT1 OR CAT2)?!
002C	61D6	IF EQ THEN	
0030	4BE6	LD R6,ACCESS_CLASS.LEVEL(R13)	!LEVEL1!
0034	5FC1	CP R6,ACCESS_CLASS.LEVEL(R14)	! COMPARE LEVEL1 WITH LEVEL2 !
0038	2101	IF GE THEN	! LEVEL1 GE LEVEL2 !
003C	5E08	LD R1,#TRUE	
0040	2101	ELSE	
0044	5E08	LD R1,#FALSE	
0048	2101	FI	
004C	5E08	ELSE	
0050	2101	LD R1,#FALSE	
0054	95F4	FI	
0058	95F2	POPL RR4,R15	
005C	9E0E	PCPL RR2,R15 !RESTORE STACK!	
0060	9E0E	RFT	
0064	9E0E	END CLASS_GE	
0068	9E0E	END NLS	
0072	9E0E	!PAGE	

APPENDIX G - SUMMARY OF REFINEMENTS

The following new procedures were added to the Inner Traffic Controller:

(1) ITC_GET_CPU_NO procedure. This procedure locates and returns the current CPU number (identification). It was used in segment management by the distributed memory manager to index into the MM_CPU_TABLE to find the memory manager process VP_ID for the current processor. This VP_ID was, in turn, used as an argument in the call to SIGNAL.

(2) ITC_GET_SEG_PTR procedure. This service procedure uses an input segment number to search the MMU_IMAGE to find the base address (pointer) for that segment. It was used in segment management to find the base address of the segment used in a process for its KST.

(3) K_LOCK and K_UNLOCK procedures. These procedures were implemented to indicate the intention to eventually have a kernel wait-lock system. K_LOCK simply calls SPIN_LOCK in its present design.

The following changes were made to the Inner Traffic Controller:

(1) The provision for a "jump table" was removed when a working version of the linker was introduced. This involved removing the constant TC_PREEMPT_HANDLER and adding an "external" declaration for the TC_PREEMPT_HANDLER Procedure.

(2) Minor changes were necessary in three procedures to modify the message size from one word to a sixteen byte array (minor changes were also needed in the declaration section). The procedures effected were: ENTER_MSG_LIST, GET_FIRST_MSG, and WAIT. The changes are documented in the code for each procedure.

The following procedures were added to the Traffic Controller:

(1) TC_GET_PROC_CLASS Procedure. This procedure locates and returns the current process's classification (i.e., it retrieves the SAC entry from the APT). It was used in segment management to retrieve the PROC_CLASS for the Segment Manager.

(3) TC_GET_DBR_NO Procedure. This procedure returns the current DBR_NO value from the APT. The Segment Manager used this procedure to obtain the DBR_NO to pass to the memory manager.

The version of the Traffic Controller shown in Appendix F is a "stub" of Reitz' [9] actual work. This stub contains the elements of the TC Module needed for proper operation of the segment management demonstration.

APPENDIX H - SEGMENT MANAGEMENT DEMONSTRATION

A. DESCRIPTION

The Seg_Mgr.Demo, as stated before, is built onto Reitz' Sync.Demo (which was designed for a different purpose than segment management, obviously). The functions illustrated by the present demonstration are: (1) virtual processor synchronization and (2) segment management function performance. The listings of the modules involved are in appendices A-F and I. It is suggested that the ASM versions be used as references; PLZ/SYS versions served as "pseudo-code" during detailed design, but are untested. The narrative discussion of the demonstration context and sequence is presented below. The output generated at each process entry point will identify the signaller in each case and the action the current process takes. The following actions are illustrated (viz., "simulated") in this demonstration. Note that this simulation uses the ITC SIGNAL/WAIT primitives instead of the TC ADVANCE/AWAIT primitives that are not yet implemented.

1. An I/C interrupt signals the IO process that a packet from the host is ready. The IO process is scheduled; it reads the packet (output: "IC: Receive Command") and signals the FM process (output: "IC: Signal FM (Create)"), passing the command (CREATE is

simulated). The IO process calls Wait, thus blocking itself while waiting for a signal from the FM process.

2. The FM process is scheduled (output: "IO=Signaller"); it interprets the command (simulated) as CREATE and thus calls CREATE_SEG in the Segment Manager (output: "FM: Call Kernel(Create)"). The normal input arguments for CREATE_SEG are passed in this call.

3. CREATE_SEG validates the CREATE request and then calls MM_CREATE_ENTRY in the Distributed Memory Manager.

4. MM_CREATE_ENTRY signals the memory manager process (MM process) and calls WAIT, thus blocking itself while waiting for a signal from the MM process.

5. The MM process is scheduled (output: "Kernel=Signaller (for FM)"); the mainline code interprets the function code and calls the CREATE_ENTRY procedure for action. When action is complete (output: "MM: CREATE_ENTRY"), the procedure returns to the mainline code. The MM process signals the return success code in a message to the FM process, then calls WAIT to wait for the next signal.

6. The FM process is scheduled (output: "Return from

Kernel"). The FM process then signals completion of CREATE to the IO process and calls WAIT (output: "FM: Signal IO").

7. The IO process is scheduled (output: "FM=Signaller"). It signals the FM process to cause a "read" to occur, i.e., the same pattern as in steps b. through e. occur for MAKE_KNOWN and SWAP_IN prior to the FM process signalling back to the IO process that the "read" was completed. This "read" is defined strictly for this test and is not equivalent to a typical Read_File packet.

8. The IO process will then be again scheduled and will perform the same functions as did the FM process (i.e., will call MAKE_KNOWN and SM_SWAP_IN sequentially) for the same segment.

9. The IO process will again be scheduled and will signal the FM process to perform the same sequence as described in g. for SM_SWAP_OUT and TERMINATE.

10. The IO process will again be scheduled and will repeat step h. with SM_SWAP_OUT and TERMINATE.

11. The IO process will again be scheduled and will cause the FM process to repeat steps b. through e. to delete (DELETE_SEG called) the segment.

12. The entire loop repeats forever.

E. INITIALIZATION

The description of the initialization of the databases is presented in figures containing the appropriate memory data. Reference to the previous descriptions of these databases and the type declarations will be useful. Figure 9 is the initialized Active Process Table. Figure 10 is the initialized Virtual Processor Table. Figure 11 is partial representation of the initialized KST for the FM process (9300) and the IO process (9700). Figure 12 is partial representation of the initialized MMU_IMAGE. Figure 13 is partial representation of the initialized process stack segments. Figure 14 is the corresponding link command line and response, and the Imager command line and response. Figure 15 is the load command lines and response, and the register initializations. Figure 16 is the output (as displayed on the CRT screen) generated by the demonstration.

APPENDIX I - DEMONSTRATION LISTINGS

INNER TRAFFIC CONTROL MODULE

```

10 1. GETWORK:
11 A. NORMAL ENTRY DOES NOT SAVE REGISTERS.
12 ( THIS IS A FUNCTION OF THE GATEKEEPER ).
13 C. R14 IS AN INPUT PARAMETER TO GETWORK THAT
14 SIMULATES INFO THAT WILL EVENTUALLY BE ON
15 THE MMU HARDWARE. THIS REGISTER MUST BE
16 ESTABLISHED AS A DBR BY ANY PROCEDURE
17 INVOKING GETWORK.
18 D. PREEMPT INTERRUPT ENTRY HANDLER, WHICH IS
19 CONTAINED IN GETWORK, DOES NOT USE THE
20 GATEKEEPER AND MUST PERFORM FUNCTIONS
21 NORMALLY ACCOMPLISHED BY IT
22 PRIOR TO NORMAL ENTRY AND EXIT.
23 ( SAVE/RESTORE: REGS, NSP; UNLOCK VPT,
24 TEST INT)
25
26 2. GENERAL:
27 A. ALL VIOLATIONS OF
28 VIRTUAL MACHINE INSTRUCTIONS ARE CONSIDERED
29 ERROR CONDITIONS AND WILL RETURN SYSTEM TO
30 MONITOR WITH ERROR CODE IN R0 AND PC IN R1.
31 B. ITC PROCEDURES CALLING GETWORK PASS DBR
32 (REGISTER R14) AS INPUT PARAMETER.
33 ( INCLUDES: SIGNAL, WAIT, SWAP, VDP, AND
34 IDLE).

```

1 PAGE

```

35 CONSTANT
36 ! ***** ERROR CODES ***** !
37 UNAUTH_LOCK := 0
38 MSG_LIST_EMPTY := 1
39 MSG_LIST_ERROR := 2
40 READY_LIST_EMPTY := 3
41 MSG_LIST_OVERFLOW := 4
42 SWAP_NOT_ALLOWED := 5
43 VP_INDEX_ERROR := 6
44
45
46 ! ***** SYSTEM PARAMETERS ***** !
47 NR_MMU_RFG := 64 ! LONG WORDS !
48 NR_VP := 4
49 IDLE_VP := NR_VP-1
50 STACK_SEG := 1
51 STACK_SEG_SIZE := %100
52 ! * * * * * OFFSETS IN STACK SEG * * * !
53 STACK_BASE := STACK_SEG_SIZE-%40
54 STATUS_REG_BLOCK := STACK_SEG_SIZE-%40
55 F_C_W := STACK_SEG_SIZE-%20
56 PROCESS_ID := STACK_SEG_SIZE-%1E
57 N_S_P := STACK_SEG_SIZE-%1C
58 ON := %FFFF
59 OFF := 0
60 RUNNING := 0
61 READY := 1
62 WAITING := 2
63 NIL := %FFFF
64 INVALID := %FFFF
65 MONITOR := %A920 ! HBUG ENTRY !
66 ! PAGE

```

```

67  TYPE
68  MESSAGE
69  ADDRESS
70  VP_INDEX
71  MSG_INDEX
72
73  MMU_TABLE_RECORD [ BASE ADDRESS
74  ATTRIBUTES WORD
75  ]
76  MSG_TABLE_RECORD
77  [ MSG
78  SENDER
79  NEXT_MSG
80  FILLER
81  ]
82
83  VP_TABLE_RECORD
84  { DRR
85  PRI
86  STATE
87  IDLE_FLAG
88  PREEMPT
89  PHYS_PROCESSOR WORD
90  NEXT_READY_VP VP_INDEX
91  MSG_LIST
92  FILLER_1
93  ARRAY [8, WORD]
94  ]
95
96  EXTERNAL
97  TC_PREEMPT_HANDLER
98  PROCEDURE
99  !PAGE

```

!

0000

```
99      INTERNAL
100      $SECTION ITC DATA
101      VPT
102      [ LOCK
103      RUNNING_LIST
104      READY_LIST
105      FREE_LIST
106      FILLER_2
107      VP
108      MSG_3
109      ]
110      IPAGE
```

WORD
VP_INDEX
VP_INDEX
MSG_INDEX
ARRAY [4, WORD]
ARRAY [NR_VP, VP_TABLE]
ARRAY [NR_VP, MSG_TABLE]

111	SECTION ITC INT PROC	
112	GETWORE	
113	*****	
114	! SWAPS VIRTUAL PROCESSORS	
115	! ON PHYSICAL PROCESSOR.	
116	*****	
117	! REGISTER USE:	
118	! STATUS REGISTERS	
119	! R0: INTERRUPT RETURN FLAG	
120	! R14: DER (SIMULATION)	
121	! R15: STACK POINTER	
122	! LOCAL VARIABLES:	
123	! R1: READY_VP (NEW)	
124	! R2: CURRENT_VP (OLD)	
125	! R3: FLAG CONTROL WORD	
126	! R4: STACK_SEG BASE ADDR	
127	! R5: STATUS_REG_BLOCK ADDR	
128	! R6: NORMAL_STACK_POINTER	
129	*****	
130	ENTRY	
131	! TURN OFF PREEMPT RETURN FLAG	
132	LD R0, #OFF	0000 2100 0000
133		
134	! GET STACK BASE	
135	LD R4, R14(#STACK_SEG*4)	0004 31E4 0004
136	LDA R5, R4(#STATUS_REG_BLOCK)	0008 3445 00C0
137		
138	! SKIP PREEMPT HANDLER	
139	JR END_PFEEMPT_HANDLER	000C E217
140		
141	PREEMPT_ENTRY: ! GLOEAL LABEL	
142	! * * PREEMPT_HANDLER * *	
143		
144	! PAGE	

145	!	SET DRR !	
146	000E	61E2	0002
147	0012	612E	0010
148			
149	0016	4D25	0014
150	001A	0001	
151			
152	001C	030F	0020
153	0020	1CF9	010F
154			
155	0024	7D67	
156	0026	93F6	
157			
158			
159			
160			
161			
162			
163			
164			
165			
166			
167	0028	31E4	0004
168	002C	3445	0000
169			
170			
171	0034	1C51	0001
172	003A	93F7	
173	0036	93F8	
174			
175			
176			
177			

```

! SET DRR !
LD R2, VPT.RUNNING LIST
LD R14, VPT.VP.DRR(R2)

! PUT CURRENT PROCESS IN READY STATE !
LD VPT.VP.READY(R2), #READY

! SAVE ALL REGISTERS !
SUB R15, #32
LDM @R15, R1, #16

! SAVE NORMAL STACK POINTER (NSP) !
LDCTL R6, NSP
PUSH @R15, R6

! SAVE LAST STATUS REGS !
!NOTE: SINCE PROCESSES CAN BE PREEMPTED ANYWHERE
IT IS NECESSARY TO HANDLE RECURSIVE CALLS
TO GETWORK. BY SAVING THE MOST RECENT SP
AND IRET FLAGS (R15 & R6) ON THE STACK
THE CONTEXT OF THESE STATUS REGISTERS IS
MAINTAINED TO ANY DEPTH OF RECURSION. !

! GET STACK PASE !
LD R4, R14(#STACK_SEG*4)
LDA R5, R4(#STATUS_REG_BLOCK)

! SAVE LAST STATUS REGS !
LDM R7, @R5, #2
PUSH @R15, R7
PUSH @R15, R8

```

! PAGE

177	!	SET INTERRUPT RETURN FLAG !
178	LD	R0, #0N
179	! *	* * * * *
180	END	PREEMPT_HANDLER:
181		
182	!	GET READY_VP LIST !
183	LD	R1, VPT.READY_LIST
184		
185	SELECT	VP:
186	DO	! UNTIL ELIGIBLE READY_VP FOUND !
187		
188	CP	VPT.VP.IDLE_FLAG(R1), #0N
189	IF	EQ ! VP IS IDLE ! THEN
190	CP	VPT.VP.PREEMPT(R1), #0N
191	IF	EQ ! PREEMPT INTERRUPT IS ON ! THEN
192	EXIT	FROM SELECT_VP
193	FI	
194	ELSE	! VP NOT IDLE !
195	EXIT	FROM SELECT_VP
196	FI	
197		
198	!	GET NEXT READY_VP !
199	LD	R3, VPT.VP.NEXT_READY_VP(R1)
200	LD	R1, R3
201	OD	
202		
203	!	NOTE: THE READY_LIST WILL NEVER BE EMPTY SINCE
204		THE IDLE_VP, WHICH IS THE LCWIST PRI_VP,
205		WILL NEVER BE REMOVED FROM THE LIST.
206		IT WILL RUN ONLY IF ALL OTHER READY_VP'S ARE
207		IDLING OR IF THERE ARE NO OTHER VP'S ON
208		THE READY_LIST. ONCE SCHEDULED, IT
209		WILL RUN UNTIL RECEIVING A HDWE INTERRUPT. !
210		
211		
212		!PAGE

211		! NOTE: R14 IS USED AS DBR HERE. WHEN MMU
212		IS AVAILABLE THIS SERIES OF SAVE AND LOAD
213		INSTRUCTIONS WILL BE REPLACED BY SPECIAL I/O
214		INSTRUCTIONS TO THE MMU. !
215		! * * SAVE SP AND INTERRUPT RETURN FLAG * * !
216	006E 1C59 0F01	LDM GR5, R15, #2
217		
218		! * * SAVE FCW * * !
219	006C 7D32	LDCTL R3, FCW
220	006E 3343 00E0	LD R4(#F_C_W), R3
221		
222	0072 4D15 0014	! PLACE NEW VP IN RUNNING STATE !
223	0076 0000	LD VPT.VP.STATE(P1), #RUNNING
224	0078 6F01 0002	LD VPT.RUNNING_LIST, R1
225		
226		! * * SWAP DBR * * !
227	007C 611E 0010	LD R14, VPT.VP.DBR(R1)
228		
229		! LOAD NEW VP SP & INTERRUPT RET FLAG !
230	0050 31E4 0004	LD R4, R14(#STACK_SEG*4)
231	0084 3445 00C0	LDA R5, R4(#STATUS_REG_BLOCK)
232	048E 1C51 0F01	LDM R15, GR5, #2
233		
234		! * * LOAD NEW FCW * * !
235	00EC 3143 00E0	LD R3, R4(#F_C_W)
236	0090 7D3A	LDCTL FCW, R3
237		
238		! TEST FOR HARDWARE INTERRUPT !
239	0092 0P00 FFFF	CP R0, #0N
240	009C 5E0E 00BC	IF EQ ! PREEMPT RETURN ! THEN
241		! HARDWARE PREEMPT INTERRUPT RETURN !
242		
243		! UNLOCK VPT !
244	009A 4DC8 0000	CLR VPT.LOCK
245		! PAGE

246	009E 5F00 0106	! TEST FOR PREEMPT !
247		! NOTE: SINCE A HW INTERRUPT DOES NOT EXIT
248		THROUGH THE GATE, THOSE FUNCTIONS PROVIDED
249		BY A GATE EXIT TO HANDLE PREEMPTS MUST BE
250		PROVIDED HERE ALSO. !
251		CALL TEST_PREEMPT
252		
253		! RESTORE LAST STATUS REGS !
254	00A2 97F6	POP R6, GR15
255	00A4 97F7	POP R7, GR15
256	00A6 1C59 0701	LDM GR5, R7, #2
257		
258	00AA 97F6	! RESTORE NSP !
259	00AC 7D6F	POP R6, GR15
260		LDC TL NSP, F6
261		! RESTORE ALL REGISTERS !
262	00AF 1CF1 010F	LDM R1, GR15, #16
263	00B2 010F 0020	ADD R15, #32
264		
265		! EXECUTE HARDWARE INTERRUPT RETURN !
266	00B6 7B02	IRET
267		
268	00FE 5F0E 00EE	ELSE ! NORMAL RETURN !
269	00BC 9E26	RET
270		FI
271	00PF	END GETWORK
272		!PAGE

273	!	00RE	ENTER MSG LIST PROCEDURE	!*****!
274	!		! INSERTS POINTER TO MESSAGE	!*****!
275	!		! FROM CURRENT VP TO SIGNED_VP	!*****!
276	!		! IN FIFO MSG LIST	!*****!
277	!		! REGISTER USE:	!*****!
278	!		! PARAMETERS:	!*****!
279	!		! R8(R9):MSG (INPUT)	!*****!
280	!		! R1: SIGNED VP (INPUT)	!*****!
281	!		! LOCAL VARIABLES:	!*****!
282	!		! R2: CURRENT_VP	!*****!
283	!		! R3: FIRST_FREE_MSG	!*****!
284	!		! R4: NEXT_FREE_MSG	!*****!
285	!		! R5: NEXT_Q_MSG	!*****!
286	!		! R6: PRESENT_Q_MSG	!*****!
287	!		!*****!	!*****!
288	!		ENTRY	!*****!
289	!		LD R2, VPT.RUNNING_LIST	!*****!
290	!			!*****!
291	!	00BE 6102 0002	! GET FIRST MSG FROM FREE LIST	!*****!
292	!		LD R3, VPT.FREE_LIST	!*****!
293	!	00C2 6103 0006	! * * * * DEFUG * * * *	!*****!
294	!		CP R3, #NIL	!*****!
295	!		IF EQ THEN	!*****!
296	!	00C6 0B03 FFFF	ILA R1, \$	!*****!
297	!	00CA 5F0F 00DA	LD R0, #MSG_LIST_OVERFLOW	!*****!
298	!	00CE 7021 00CE	CALL MONITOR	!*****!
299	!	00D2 2100 0004	FI	!*****!
300	!	00D6 5F00 A900	! * * * * END DEBUG * * * *	!*****!
301	!			!*****!
302	!			!*****!
303	!			!*****!
304	!			!*****!
305	!	00DA 6134 20A2	R4, VPT.MSG.Q.NEXT_MSG(R3)	!*****!
306	!	00DE 6F04 0000	VPT.FREE_LIST, R4	!*****!
307	!			!*****!

IPAGE

!			302
00E2 7C3A 0090'	! INSERT MESSAGE LIST INFORMATION !		309
00E6 2107 0010	LDA R10,VPT.MSG Q.MSG(R3)		310
00EA B8B1 07A0	LD R7,#SIZECF MESSAGE		311
00EE 6F32 40A0	LDIRE @R10,@R6,R7		312
	LD VPT.MSG Q.SENDER(R3), R2		313
	! INSERT MSG IN MSG LIST !		314
00F2 6115 001E'	LD R5, VPT.VP.MSG LIST(R1)		315
			316
00F6 0R05 FFFF	CP R5, #NIL		317
00FA 5F0E 4106'	IF EQ ! MSG LIST IS EMPTY ! THEN		318
	! INSERT MSG AT TOP OF LIST !		319
00FE 6F13 001E'	LD VPT.VP.MSG LIST(R1), R3		320
			321
0102 5F08 011E'	ELSE ! INSERT MSG IN LIST !		322
	MSG Q_SEARCH:		323
	DO ! WHILE NOT END OF LIST !		324
0106 0R05 FFFF	CP R5, #NIL		325
010A 5E0E 0112'	IF EQ ! END OF LIST ! THEN		326
010F 5F08 011A'	EXIT FROM MSG_Q_SEARCH		327
	FI		328
			329
	! GET NEXT LINK !		330
0112 A156	LD R6, R5		331
0114 61C5 00A2'	LD R5, VPT.MSG Q.NEXT_MSG(R6)		332
0116 E8F6	OD		333
	! INSERT MSG IN LIST !		334
011A 6F63 00A2'	LD VPT.MSG_Q.NEXT_MSG(R6), R3		335
	FI		336
011E 6F35 00A2'	LD VPT.MSG_Q.NEXT_MSG(R3), R5		337
0122 9E08	RET		338
0124	END ENTER MSG_LIST		339
			340 !PAGE

! 0124

```

341 GET_FIRST_MSG PROCEDURE
342 !*****
343 ! REMOVES MSG FROM MSG_LIST
344 ! AND PLACES ON FREE_LIST.
345 ! RETURNS SENDER'S MSG ANL
346 ! VP_ID
347 !*****
348 REGISTER USE:
349 ! PARAMETERS:
350 ! R8(R9): MSG POINTER (INPUT)
351 ! R1: SENDER VP (RETURNED)
352 ! LOCAL VARIABLES
353 ! R2: CURRENT VP
354 ! R3: FIRST_MSG
355 ! R4: NEXT_MSG
356 ! R5: NEXT_FREE_MSG
357 ! R6: PRESENT_FREE_MSG
358 !*****
359 ENTRY
360 LD R2, VPT.RUNNING_LIST
361
362 ! REMOVE FIRST MSG FROM MSG_LIST !
363 LD R3, VPT.VP.MSG_LIST(R2)
364
365 ! * * * * * DEBUG * * * * !
366 CP R3, #NIL
367 IF EQ THEN
368 LD R2, #MSG_LIST_EMPTY
369 LDA R1, 5
370 CALL MONITOR
371 FI
372 ! * * * * * END DEBUG * * * * !
373 R4, VPT.MSG_V.NEXT_MSG(R3)
374 VPT.VP.MSG_LIST(R2), R4
375
376 LD
377 LD

```

! PAGE

!				376	! INSERT MESSAGE IN FREE LIST !
0148 6135 0006'				377	LD R5, VPT.FREE LIST
014C 0F05 FFFF'				378	CP R5, #NIL
2150 5E0F 0162'				379	IF EQ ! FREE_LIST IS EMPTY ! THEN
				380	! INSERT AT TOP OF LIST !
0154 6F03 0006'				381	LD VPT.FREE_LIST, R3
0158 4D35 00A2'				382	LD VPT.MSG_Q.NEXT_MSG(R3), #NIL
015C FFFF'					
015E 5E05 017F'				383	ELSE ! INSERT IN LIST !
				384	FREE_Q_SEARCH:
				385	DO
				386	
0162 0B05 FFFF'				387	CP R5, #NIL
0166 5E0F 0161'				388	IF EQ ! END OF LIST ! THEN
016A 5E28 0176'				389	EXIT FROM FREE_Q_SEARCH
				390	FI
				391	! GET NEXT MSG !
016F A156				392	LD R6, R5
0170 6165 00A2'				393	LD R5, VPT.MSG_Q.NEXT_MSG(R6)
0174 18F6				394	OD
				395	
0176 6F63 00A2'				396	! INSERT IN LIST !
017A 6135 00A2'				397	LD VPT.MSG_Q.NEXT_MSG(R6), R3
				398	LD VPT.MSG_Q.NEXT_MSG(R3), R5
				399	FI
				400	! GET MESSAGE INFORMATION:
				401	(RETURNS R1: SENDING VP) !
017E 6131 00A0'				402	LD R1, VPT.MSG_Q.SENDER(R3)
0182 763A 0090'				403	LDA R10, VPT.MSG_Q.MSG(R3)
0186 2107 0210				404	LD R7, #SIZEOF MESSAGE
018A 3A11 0780				405	LDIRB QRR, QRI0, R7
018F 9F05				406	RET
0190				407	END GET_FIRST_MSG
				408	

! 0190

```

409 MAKE_READY PROCEDURE
410 !*****
411 ! INSERTS SCHEDULE VP ID INTO !
412 ! READY LIST IAW PRIORITY AND !
413 ! PUTS IT IN READY STATE. !
414 !*****
415 ! REGISTER USE: !
416 ! PARAMETERS: !
417 ! R1: SIGNALFD VP (INPUT) !
418 ! LOCAL VARIABLES !
419 ! R2: SIG VP.PRI !
420 ! R3: PRESENT_VP !
421 ! R4: NEXT_VP !
422 !*****
423 ENTRY
424 IL
425 ! * * * DERUG * * * !
426 CP R4, #NIL
427 IF EQ ! LIST IS EMPTY ! THEN
428 LD R0, #READY_LIST_EMPTY
429 LEA R1, 5
430 CALL MONITOR
431 FI
432 ! * * * INL DEUG * * * !
433
434 LD R2, VPT.VP.PRI (R1)
435
436 CP R2, VPT.VP.PRI(R4)
437 IF GT ! SIG VP.PRI > READY VP.PRI ! THEN
438 ! INSERT AT FRONT OF LIST !
439 LD VPT.VP.NEXT_READY_VP(R1), R4
440 LD VPT.READY_LIST, R1
441 !PAGE

```

0190 6104 0002
0194 0F04 FFFF
0198 5E0E 01A2
019C 2100 0003
01A0 7601 01A0
01A4 5F00 A900

01A8 6112 0012
01AC 4B42 0012
01F0 5E02 01C0
01B4 6F14 001C
01P8 6F01 0004

!	01RC 5E08 01E8'	442	ELSE ! INSERT IN LIST !
		443	
		444	READY LIST SEARCE:
01C0 0E04 FFFF		445	DO ! WHILE NOT END OF LIST !
01C4 5E0E 01CC'		446	CP R4, #NIL
01C8 5E08 01E0'		447	IF FQ ! IF END OF LIST ! THEN
		448	EXIT FROM READY LIST SEARCE
		449	FI
		450	
01C0 4B42 0012'		451	CP R2, VPT.VP.PRI (R4)
01D0 5E02 01D8'		452	IF GT ! SIG VP.PRI > PRESENT VP.PRI ! THEN
01D4 5E0E 01E0'		453	EXIT FROM READY LIST SEARCE
		454	FI
		455	
		456	! GET NEXT LINK !
01D8 A143		457	LD R3, R4
01DA 6134 001C'		458	LD R4, VPT.VP.NEXT_READY VP(R3)
01DE E2F0		459	
		460	CD
01E0 6F14 001C'		461	! INSERT SIG VP IN LIST !
01E4 6F31 001C'		462	LD VPT.VP.NEXT_READY VP(R1), R4
		463	LD VPT.VP.NEXT_READY VP(R3), R1
		464	FI
		465	
01F0 4D15 0014'		466	! CHANGE STATE TO READY !
01EC 0021		467	LD VPT.VP.STATE(R1), #READY
		468	
01FF 9E0E		469	RET
01F0		470	END MAKE_READY
		471	IPAGE

```

472 ! * * INNER TRAFFIC CONTROL ENTRY POINTS * * !
473 GLOBAL
474 %SECTION ITC_GLB_PROC
475
476 EARDWARE_PHEMPT LAEFL
477
478 WAIT
479 ! ***** PROCEDURE ***** !
480 ! INTRA_KERNEL SYNC/COM PRIMITIVE !
481 ! INVOKED BY KERNEL PROCESSES !
482 ! ***** !
483 ! PARAMETERS !
484 ! R0(R9): MSG POINTER (INPUT) !
485 ! R1: SENDING VP (RETURN) !
486 ! GLOBAL VARIABLES !
487 ! R14: DFR (PARAM TO GETWORK) !
488 ! LOCAL VARIABLES !
489 ! R2: CURRENT VP (RUNNING) !
490 ! R3: NEXT_READY VP !
491 ! R4: LOCK_ADDRESS !
492 ! ***** !
493 ENTRY
494 ! LOCK VPT !
495 LDA R4, VPT_LOCK
496 CALL SPIN_LOCK ! (R4: VPT_LOCK) !
497 ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
498
499 LD R2, VPT.RUNNING_LIST
500 LD R3, VPT.VP.NEXT_READY_VP(R2)
501 CP VPT.VP.MSG_LIST(R2), #NIL
502
503 IF EQ ! CURRENT VP'S MSG LIST IS EMPTY ! THEN
504 ! REMOVE CURRENT_VP FROM READY LIST !
505 ! PAGE

```

!		536	! * * * * * DEBUG * * * * *
001A 0103	FFFF	507	CP R3, #NIL
001E 5E0E	002E	508	IF EQ THEN
0022 2100	0003	509	LD R0, #READY_LIST_EMPTY
0026 7601	0026	510	ILA R1, 5
002A 5F02	A900	511	CALL MONITOR
		512	FI
		513	! * * * * * END DEBUG * * * * *
		514	
002E 6F03	0004	515	LD VPT.READY_LIST, R3
0032 4D25	0010	516	LD VPT.VP.NEXT_READY_VP(R2), #NIL
0036 FFFF			
		517	
0038 4D25	0014	518	! PUT IT IN WAITING STATE !
003C 0002		519	LD VPT.VP.STATE(R2), #WAITING
003E 612E	0010	520	! SET DBR !
		521	LD R14, VPT.VP.DBR(R2)
0042 93FE		522	! SCHEDULE FIRST ELIGIBLE READY VP !
0044 5F02	0000	523	PUSH GR15, R8 !SAVE MSG POINTER!
0048 97FE		524	CALL GETWORK ! (R14: DBR) !
		525	PCP R8, GR15
		526	
		527	FI
004A 5F00	0124	528	! GET FIRST MSG ON CURRENT VP'S MSG LIST !
		529	CALL GET_FIRST_MSG !COPIES MSG IN MSG ARRAY!
		530	!RETURNS R1:SENDER_VP !
		531	
004E 4D0E	0000	532	! UNLOCK VPT !
		533	CLR VPT.LOCK
		534	
0052 9F26		535	! RETURN: R1:SENDER_VP !
0054		536	RET
		537	END WAIT
		538	
		539	!PAGE

! 0054

```

540 SIGNAL      PROCEDURE
541 !*****
542 ! INTRA_KERNEL SYNC /COM PRIMITIVE !
543 ! INVOKED BY KERNEL PROCESSES
544 !*****
545 ! REGISTER USE:
546 !
547 ! PARAMETERS:
548 !   RE(R9): MSG POINTER (INPUT)
549 !   R1: SIGNED VP ID (INPUT)
550 ! GLOBAL VARIABLES
551 !   R14: DPH (PARAM TO GETWORK)
552 ! LOCAL VARIABLES:
553 !   R1: SIGNED VP
554 !   R2: CURRENT VP
555 !   R4: VPT.LOCK ADDRESS
556 !*****
557 ENTRY
558 ! LOCK VPT !
559 LDA     R4, VPT.LOCK
560 CALL    SPIN_LOCK ! (R4: VPT.LOCK) !
561 ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !
562 ! PLACE MSG IN SIGNED VP'S MSG LIST !
563 CALL ENTER_MSG_LIST ! (R2:MSG PTR,
564 !                     R1:SIGNED VP) !
565 CP      VPT.VP.STATE(R1), #WAITING
566 IF EQ   ! SIGNED VP IS WAITING ! THEN
567 ! WAKE IT UP AND MAKE IT READY !
568 CALL MAKE_READY ! (R1: SIGNED_VP) !
569 ! PUT CURRENT_VP IN READY_STATE !
570 LD      R2, VPT.RUNNING_LIST
571 LD      VPT.VP.STATE(R2), #READY
572
573
574 IPAGE

```

!			575	! SET DPR !
007E	612E	0010'	576	LD R14, VPT.VP.DRR(R2)
			577	
007C	5120	0000'	578	! SCHEDULE FIRST ELIGIBLE READY VP !
			579	CALL GETWORK !(R14: DPR) !
			580	FI
			581	
0080	4D08	0000'	582	! UNLOCK VPT !
			583	CLR VPT.LOCK
0084	9E08		584	
008C			585	RFT
			586	END SIGNAL
			587	!PAGE

0086	!	SET_PREEPT	PROCEDURE	*****!
588			! *****!	! SETS PREEPT INTERRUPT ON!
589			! *****!	! TARGET VP. CALLED BY TC.
590			! *****!	! ADVANCE.
591			! *****!	! *****!
592			! *****!	! REGISTER USE:
593			! *****!	! PARAMETERS:
594			! *****!	! R1:TARGET VP ID (INPUT)
595			! *****!	! LOCAL VARIABLES
596			! *****!	! R1: VP INDEX
597			! *****!	! *****!
598			! *****!	! *****!
599			! *****!	! *****!
600			ENTRY	
601			! NOTE: DESIGNED AS SAFE SEQUENCE SO VPT NEED	
602			NOT BE LOCKED. !	
603				
604			! CONVERT VP_ID TO VP_INDEX !	
605	0086	BD00	LDK R0, #0	
606	0088	1900	MULT RR0, #SIZEOF VP_TABLE	
607		0020	! THIS LEAVES VP_INDEX IN R1 !	
608				
609			! TURN ON TGT_VP_PREEPT_FLAG !	
610	00EC	4D15	LD VPT.VP.PREEPT(R1), #ON	
611	0090	FFFF		
612			! ** IF TARGET VP NOT LOCAL	
613			(NOT BOUND TO THIS CPU)	
614			[IE, IF <<CPU_SEG>>CPU_ID<>VPT.VP.PHYS_CPU(P1)]	
615			THEN SEND HARDWARE PREEPT INTERRUPT TO	
616			VPT.VP.CPU(R1). ** !	
617				
618	0092	9E0E	RET	
619	0094		END SET_PREEPT	
620				IPAGE

! 0004

```

621 IDLE PRCFDURE *****!
622 ! *****!
623 ! LOADS IDLE DBR ON !
624 ! CURRENT VP. CALLED BY !
625 ! TC GETWORK. !
626 ! *****!
627 ! REGISTER USE !
628 ! GLOBAL VARIABLE !
629 ! R14: DBR !
630 ! LOCAL VARIABLES: !
631 ! R2: CURRENT VP !
632 ! R3: TEMP VAR !
633 ! R4: VPT.LOCK ADDR !
634 ! R5: TEMP !
635 ! *****!
636 ENTRY
637 ! LOCK VPT !
638 LDA R4, VPT.LOCK
639 CALL SPIN LOCK ! (R4: VPT.LOCK) !
640 ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP!
641
642 ! GET CURRENT VP !
643 LD R2, VPT.RUNNING LIST
644
645 ! SET DBR !
646 LD R14, VPT.VP.DBR(R2)
647
648 ! LOAD IDLE DBR ON CURRENT VP !
649 LD R3, #IDLE_VP*SIZECF VP TABLE
650 LD R5, VPT.VP.DBR(R3)
651 LD VPT.VP.DBR(R2), R5
652
653 ! TJRN ON CURRENT VP'S IDLE FLAG !
654 LD VPT.VP.IDLE_FLAG(R2), #ON
655 !PAGE

```

!		656	! SET VP TO READY STATE !
00P6 4D25 0014'		657	LD VPT.VP.STATE(R2), #READY
00BA 0001			
		658	! SCHEDULE FIRST ELIGIBLE READY VP !
00PC 5F00 0000'		659	CALL GETWORK !(R14: DBR) !
		660	
		661	! UNLOCK VPT !
		662	CLR VPT.LOCK
00C0 4D08 0000'		663	
		664	
00C4 9F00		665	RFT
00CC		666	END IDLE
		667	IPAGE

! 00C6

```

668 SWAP VDRR PROCEDURE
669 !*****!
670 ! LOADS NEW DFR ON !
671 ! CURRENT VP. CALLED BY !
672 ! TC GETWRK. !
673 !*****!
674 ! REGISTER USE !
675 ! PARAMETERS !
676 ! R1: NEW LFR (INPUT) !
677 ! GLOBAL VARIABLES !
678 ! R14: DRR !
679 ! LOCAL VARIABLES !
680 ! R2: CURRENT VP !
681 ! R4: VPT.LOCK ADDR !
682 !*****!
683 ENTRY
684 ! LOCK VPT !
685 LDA R4, VPT.LOCK
686 CALL SPIN LOCK ! (R4: VPT.LOCK) !
687 ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !
688 ! GET CURRENT VP !
689 LD R2, VPT.RUNNING_LIST
690 ! * * * DEFUG * * * !
691 CP VPT.VP.MSG_LIST(R2), #NIL
692 IF NE ! MSG WAITING ! THEN
693 LD R0, #SWAP_NOT_ALLOWED
694 LDA R1, $ !PC!
695 CALL MONITOR
696 FI
697 ! * * * END DEBUG * * * !
698 ! SET DIR !
699 LD R14, VPT.VP.DFR(R2)
700
701 !PAGE

```

AD-A094 569

NAVAL POSTGRADUATE SCHOOL MONTEREY CA
IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A SECURE ARCHIVAL STOR-ETC(U)
SEP 80 J T WELLS

F/G 9/2

UNCLASSIFIED

NL

3 of 3

AD-A
094569



END
DATE
FILMED
3-81
DTIC

1					
00EC 6F21	0010	702	! LOAD NEW DRR ON CURRENT VP !		
		703	LD VPT.VP.DRR(R2), R1		
		704			
00F0 4D25	0016	705	! TURN OFF IDLE FLAG !		
00F4 0002		706	LD VPT.VP.IDLE_FLAG(R2), #OFF		
00F6 4D25	0014	707	! SET VP TO READY STATE !		
00FA 0201		708	LD VPT.VP.STATE(R2), #READY		
		709			
00FC 5F02	0000	710	! SCHEDULE FIRST ELIGIBLE READY VP !		
		711	CALL GETWORK ! (R14:DRR) !		
		712			
		713			
		714	! UNLOCK VPT !		
0103 4D08	0000	715	CLR VPT.LOCK		
		716			
0104 9E0E		717	RFT		
0106		718	END SWAP_VDRR		
		719			

IPAGE

! 0106

```

720
721
722
723
724
725
726
727
728
729
730
731.
732.
733
734.
735
736
737
738
739.
740.
741.
742
743
744.
745
746.
747.
748
749
750.
751.
752.
753 !PAGE

TEST_PREEMPT          PROCEDURE
!*****
! TESTS FOR PREEMPT INTERRUPT !
! FLAG AND HANDLES INTERRUPT !
! IF FLAG IS SET.            !
! INVOKED UPON EVERY EXIT FROM !
! KERNEL.                     !
!*****
! REGISTER USE                !
! LOCAL VARIABLES             !
! R1: PREEMPT_INT_FLAG        !
! R2: CURRENT_VP              !
!*****
ENTRY
TEST_FLAG:
DO ! WHILE CURRENT_VP'S PREEMPT FLAG IS ON !

!NOTE: NEXT TWO STATEMENTS MAY NOT BE RACE FREE.
LOCK MAY BE REQUIRED FOR MULTIPROCESSOR SYS!

! GET CURRENT_VP !
LD R2, VPT.RUNNING_LIST

! TEST PREEMPT INTERRUPT FLAG !
LD R1, VPT.VP.PREEMPT(R2)
CP R1, #OFF
IF EQ ! PREEMPT FLAG IS OFF ! THEN
EXIT FROM TEST_FLAG
FI

! *** VIRTUAL PREEMPT HANDLER *** !
! ** NOTE: SAFE SEQUENCE AND DOES NOT REQUIRE
VPT TO FF LOCKED. ** !

```

0106 6102 0002
010A 6121 0018
010F 0FE1 0000
0112 5E0E 011A
0116 5E06 012C

!		754	! RESET PREEMPT FLAG !
011A 4D25 2018'		755	LD VPT.VP.PREEMPT(R2), #OFF
011E 0320		756	
0120 5F00 0000*		757	! SIMULATE PREEMPT INTERRUPT !
		758	CALL TC_PREEMPT_HANDLER
		759	! NOTE: THIS JUMP TO TRAFFIC CONTROL
		760	IS USED ONLY IN THE CASE OF A PREEMPT INTERRUPT
		761	AND SIMULATES A HARDWARE INTERRUPT. ** !
		762	
		763	! *** END VIRTUAL PREEMPT HANDLER *** !
0124 E8F0		764	OD
		765	
012E 9E08		766	! RETURN TO GATEKEEPER !
		767	RET
012E		768	
		769	END TEST PREEMPT
		770	!PAGE

```

;
! 012E
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
! PAGE

!
012E 7604 0000
012C 5F00 0154
0130 6101 0002
0134 FD00
0136 1P04 0020
013A 0B00 0000
013F 5F06 014F
0142 2100 0006
014C 7601 0146
014A 5F06 A900
014F 410E 0000
0152 0E0E
0154

```

```

RUNNING VP PROCEDURE
!*****!
! CALLED BY TRAFFIC CONTROL.
! RETURNS VP ID. RESULT IS VALID
! ONLY WHILE APT IS LOCKED.
!*****!
! REGISTER USE
! PARAMETERS
! R1: VP ID (RETURNED)
! LOCAL VARIABLES
! R0: DIVIDEND
! R0: REMAINDER
! R1: QUOTIENT
!*****!
ENTRY
! LOCK VPT !
LDA R4, VPT.LOCK
CALL SPIN_LOCK ! (R4: VPT.LOCK) !
! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
LD R1, VPT.RUNNING_LIST
LDK R0, #0
! CONVERT VP_INDEX TO VP_ID !
DIV R0, #SIZEOF_VPTABLE
! * * * DEBUG * * * !
CP R0, #0
IF NE, IREMAINDER <> 0 ! THEN
LD R0, #VP_INDEX_ERROR
LDA R1, $
CALL MCNITOR
FI
! * * * END DEBUG * * * !
CLR VPT.LOCK
RET
END RUNNING_VP

```

! 0154

806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839

0154 0D41 0002
0158 5E06 0168
015C 2100 0000
0160 7601 0160
0164 5F00 A900

0168 0D40
016A 1E0E

016C 9E00

016E

IPAGE

```

SPIN_LOCK PROCEDURE
!*****
!  USES SPIN_LOCK MECH.
!  LOCKS UNLOCKED DATA
!  STRUCTURE (POINTED TO)
!  BY INPUT PARAMETER.
!*****
!REGISTER USE
!
!PARAMETERS
!  R4: LOCK ADDR (INPUT)!
!*****
ENTRY
! NCIE: SINCE ONLY ONE PROCESSOR CURRENTLY IN
!       SYSTEM, LOCK NOT NECESSARY. **
!
! ** DFUG **
CP @R4, #OFF
IF NE ! NOT UNLOCKED ! THEN
LD R0, #UNLOCKED_LOCK
LDA R1, $
CALL MONITOR
FI
! ** END DEBUG **
!
TEST_LOCK:
!  DO WHILE STRUCTURE LOCKED !
TSET @R4
JR M1, TFST_LOCK
! ** NOTE SEE PLZ/ASM MANUAL
!       FOR RESTRICTIONS ON
!       USE OF TSET. **
RET
END SPIN_LOCK

```

```

      840 ITC_GET_CPU_NO ***** PKCEDURE
      841 ***** !
      842 ! IIND CURRENT CPU_NO !
      843 ! CALLED BY DIST MMGR !
      844 ***** !
      845 ! REGISTER USE !
      846 ! R1: CPU_NO (RETURNED) !
      847 ! R2: VP INDEX (LCCAL) !
      848 ***** !
      849
      850 ENTRY
      851 LDA R4,VPT.LOCK
      852 CALL SPIN LOCK ! R4: ~VPT.LOCK !
      853 LD R2,VPT.RUNNING_LIST
      854 LD R1,VPT.VP.PHYS_PROCESSOR(R2)
      855 CLR VPT.LOCK
      856 RET
      857 END ITC_GET_CPU_NO
      858
      859
      860 ! PAGE

```

! 0184

```

861 ITC GET_SEG_PTR          PROCEDURE
862 !*****!
863 ! JFS BASE ADDRESS OF SEGMENT !
864 ! INDICATED. !
865 !*****!
866 ! REGISTER USE: !
867 ! P0:SEG BASE ADDRESS(RET) !
868 ! R1:SEG NR (INPUT) !
869 ! R2:RUNNING_VP (LOCAL) !
870 ! P3:DER_VALUE (LOCAL) !
871 ! R4:VPT_LOCK !
872 !*****!
873
874 ENTRY
875 LDA R4,VPT_LOCK
876 CALL SPIN_LOCK !R4:VPT_LOCK!
877 LD R2,VPT.RUNNING_LIST
878 LD R3,VPT.VP.DBR(R2)
879 CLR VPT_LOCK
880 MULT RPJ,#4
881 LD R6,R3(R1) INOTE: DER (NOT DER NO)
882 USED HERE!
883 RET
884 END ITC_GET_SEG_PTR
885 !PAGE

```

0184 7644 0000
018E 5F00 0154
018C 6102 0002
0190 6123 0010
0194 4D08 0000
0198 1900 0004
019C 7130 0100
01A0 9F38
01A2

```

!
01A2
!
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
!PAGE

01A2 5F00 0154
01A6 9E08
01A8

01A8

01A8 0L4E
01AA 9E0E
01AC

K_LOCK
PROCEDURE
!*****!
! STUB FOR WAIT LOCK!
!*****!
! R4: LOCK (INPUT) !
!*****!

ENTRY
CALL SPIN_LOCK
RET
END K_LOCK

K_UNLOCK
PROCEDURE
!*****!
! STUB FOR WAIT UNLOCK !
!*****!
! R4: LOCK (INPUT) !
!*****!

ENTRY
CLR QP4
RET
END K_UNLOCK

END INNER_TRAFFIC_CONTROL

```


36	TYPE	
37	AP_POINTER	WORD
38	ADDRESS	WORD
39	EVENT_TABLE	RECORD
40	[HANDLE	WORD
41	EVENT	WORD
42	TICKET	WORD
43	FILLER_2	ARRAY [5 WORD]
44	]	
45	AP_TABLE	RECORD
46	[DER_NO	SHORT_INTEGER
47	FILLER_0	BYTE
48	SAC	LONG
49	PRI	INTEGER
50	STATE	INTEGER
51	NEXT_AP	AP_POINTER
52	FILLER_1	ARRAY [2 WORD]
53	EVENTCOUNT	EVENT_TABLE
54	]	
55		
56	RUNNING_ARRAY	ARRAY [NR_AVAILABLE VP WORD]
57	IPAGE	

```

58  EXTERNAL
59  SPIN LOCK
60  SET_PREFMPT
61  SWAP_VDBR
62  IDLE
63  RUNNING_VP
64  MM_GET_DBR_VALUE
65  K LOCK
66  K UNLOCK
67
68  $SECTION TC_DATA
69  INTERNAL
70  APT RECORD
71  [ SUCCESS_CODE
72  LOCK
73  RUNNING_LIST
74  READY_LIST
75  BLOCKED_LIST
76  FILLER
77  AP
78  )
79  )

```

2222

GLCEAL
SECTION TC_GLB_PROC

```

0000 TC GETWORK PROCEDURE
0001 ! *****!
0002 ! LOADS NEXT READY DBR !
0003 ! ON CURRENT VP. !
0004 ! *****!
0005 ! REGISTER USE !
0006 ! LOCAL VARS !
0007 ! K1: CURRENT_VP_ID !
0008 ! R2: READY_AP !
0009 ! R3: VP_PTR !
0010 ! *****!
0011 ENTRY
0012 ! FIND FIRST READY PROCESSOR !
0013 ID R2, APT.READY_LIST
0014 READY_AP_SEARCH:
0015 LC ! WHILE NOT (END OF LIST OR READY PROCESS) !
0016
0017 CP R2, #NIL
0018 IF EQ ! IF NO READY PROCESSES ! THEN
0019 EXIT FROM READY_AP_SEARCH
0020 FI
0021
0022 CP APT.AP.STATUS(R2), #READY
0023
0024 IF EQ ! IF PROCESS READY ! THEN
0025 EXIT FROM READY_AP_SEARCH
0026 FI
0027
0028 ! GET NEXT READY AP !
0029 LD R3, APT.AP.NEXT_AP(R2)
0030 LL R2, R3
0031 OD
0032
0033 ! PAGE

```

0026	0102	FFFF	115	CP	R2,#NIL
002A	5E0F	003C	116	IF EQ !	IF NO PROCESSES READY ! THEN
002F	4D15	0004	117	! LOAD	IDLE PROCESS !
0032	DDDD		118	LD	APT.RUNNING LIST(R1), #IDLE1
0034	5F00	0000*	119	CALL	IDLE
0035	5F08	0054	120	ELSE	
003C	CF12	0004	121	! LOAD	FIRST READY AP !
0040	4D25	0018	122	LD	APT.RUNNING LIST(P1), R2
0044	0000		123	LD	APT.AP.STATE(R2), #RUNNING
0046	6029	0013	124	LDR	RL1, APT.AP.DBR_NO(F2)
004A	5F00	0000*	125	CALL	MM_GET_DBR_VALUE ! (R1:DBR_NO) !
004E	A121		126	LD	R1,R2 !DBR_VALUE ! (RETURNS:F2:IFR) !
0050	5F00	0000*	127	CALL	SWAP_VDPR ! (R1:DBR) !
0054	9E08		128	FI	
0056			129	RET	
			130	FND	TC_GETWORK
			131		
			132		!PAGE

133	TC_PREFMPT_HANDLER	PROCEDURE
134	*****	*****
135	! LOADS FIRST READY AP	*****
136	! IN RESPONSE TO PREEMPT	*****
137	! INTERRUPT	*****
138	*****	*****
139	! GLOBAL (TC) VARIABLES	*****
140	! K12: CURRENT PROCESS	*****
141	! R13: SEG_BASE_ADDP	*****
142	! R14: DEF_NO	*****
143	*****	*****
144	*****	*****
145	ENTRY	*****
146	! ** CALL WAIT LOCK (APT^.LOCK) **	*****
147	! ** RETURNS WHEN PROCESS HAS LOCKED APT **	*****
148	! GET RUNNING VP ID	*****
149	CALL RUNNING_VP (RETURNS: R1:VP_ID)	*****
150	! GET AP	*****
151	LD R2, APT.RUNNING_LIST(R1)	*****
152	! IF NOT AN IDLE PROCESS, SET IT TO READY	*****
153	CP R2, #IDLE1	*****
154	IF NE ! NOT IDLE ! THEN	*****
155	LD APT.AP.STATE(R2), #READY	*****
156	FI	*****
157	! LOAD FIRST READY PROCESS	*****
158	CALL TC_GETWORK	*****
159	! ** CALL WAIT_UNLOCK (APT^.LOCK) **	*****
160	! ** RETURNS WHEN PROCESS HAS UNLOCKED APT **	*****
161	! ** AND ADVANCED ON THIS EVENT **	*****
162	RET	*****
163	END TC_PREFMPT_HANDLER	*****
164		*****
165		*****
166		*****

! 0072

```

167 TC_GET_PROC_CLASS      PROCEDURE
168 !*****!
169 ! SIARCHIS APT FOR !
170 ! CURRENT PROC CLASS !
171 ! CALLED BY SFG MGR !
172 !*****!
173 ! REGISTER USE !
174 ! PARAMETERS !
175 ! R1:CUR VP ID(LCCAL) !
176 ! R2:SAC(RETURNS) !
177 ! R5:PROCESS ID(LOCAL)!
178 !*****!
179 !
180 ENTRY
181 LDA R4,APT.LOCK ! (R4: APT.LOCK)!
182 CALL K_LOCK ! (RETURNS R1:VP ID)!
183 CALL RUNNING_VP ! (RETURNS R1:VP ID)!
184 ID R5,APT.RUNNING_LIST(R1)
185 LDL R2,APT.AP.SAC(R5)
186 CLR APT.LOCK
187 RET
188 END TC_GET_PROC_CLASS
189 IPAGE

```

```

2272 7624 0002'
2074 5F00 0000*
207A 5F00 0000*
207E 6115 0004'
2082 5452 0012'
2086 4108 0002'
208A 9E28
208C

```

! 338C

```

190 TC GET_DBR_NO PROCEDURE
191 !*****!
192 ! SEARCHES APT FOR CURRFNT VALUE!
193 ! OF_DBR_NO. !
194 !*****!
195 ! REGISTEE USE: !
196 ! R1:CUR_VP_ID (LOCAL) !
197 ! R1:DER_NO (RETURNED) !
198 ! R1:APT_LOCK (LOCAL) !
199 ! R5:PROCESS_ID (LOCAL) !
200 !*****!
201 !*****!
202 ENTRY
203 LDA R4,APT_LOCK
204 CALL K_LOCK !R4: APT_LOCK!
205 CALL RJNNING_VP ! (RETURNS:R1:VP_ID) !
206 LD R5,APT.RUNNING_LIST(R1)
207 LDB R11,APT.AP.DER_NO(R5)
208 CLR APT_LOCK
209 RET
210 END TC_GET_DBR_NO
211
212 END TC
213
214 !PAGE

```

```

005C 76C4 0002'
0290 5F00 0002*
0094 5F00 0000*
0098 6115 0004'
029C 6059 0010'
00A0 4D08 0002'
00A4 9F08
00A6

```

```

2  IDLE_PROCESS_MODULE
3  1  VERS. 1.8 1
4  CONSTANT
5  WRITEIN      := %0FC3
6  RETURN_TO_PPUG := %A902
7
8  EXTERNAL
9  SIGNAL      PROCEDURE
10 WAIT      PROCEDURE
11 IDLE      PROCEDURE
12 SET_PALMPT PROCEDURE
13 SWAP_VDRR PROCEDURE
14 TEST_PALMPT PROCEDURE
15
16 INTERNAL
17
18 $SECTION IDLE_DATA
19 MSG ARRAY [* EYTE] := '1 ** IDLE PROCESS IS RUNNING! ** %R'

```

```

0000 49 20 2A 49
0003 2A 20 4C 45
0006 44 40 50 52
0009 20 43 53 45
000C 4F 53 53 20
000F 53 53 53 20
0012 49 53 55 4E
0015 52 55 49 4E
0018 4E 49 21 20
001F 47 2A 2A 20
0021 2A 2A 20 20

```

20 !PAGE

!			
0000			
0000	2102	0000	
0004	5F00	0FC0	
0008	5F00	A902	
000C	9F0E		
000E	0B0A	0000	
0012	5F0E	001F	
0016	5F00	0000*	
001A	5E08	0052	
001F	0E0A	0001	
0022	5E0F	002F	
0026	5F00	0000*	
002A	5E0E	0052	
002F	0B0A	0002	
0032	5E0E	003E	
0036	5F00	0000*	
003A	5E0E	0052	
003E	0B0A	0003	
0042	5F0E	004F	
0046	5F00	0000*	
004A	5E08	0052	
004E	5F0E	A902	
0052	9E05		
0054			

```

21 $SECTION IDLE_PROC
22 IDLE_MAIN PROCEDURE
23
24 ENTRY
25 LD R2, #MSG
26 CALL WRITEIN
27 CALL RETURN_TO_HRUG
28 RET
29
30 TEST ENTRY:
31 ! PARAMETER R10: CALLS PROCEDURES !
32 IF R10
33 CASE #0 THEN CALL IDLE
34
35 CASE #1 THEN CALL SWAP_VDHR
36
37 CASE #2 THEN CALL SET_PREEMPT
38
39 CASE #3 THEN CALL TEST_PREEMPT
40
41 ELSE
42 CALL RETURN_TO_HRUG
43
44 FI
45
46 RET
47
48 END IDLE_MAIN
49
50 FND ILLF_PROCESS
51
52 !PAGE

```

```

2 MM_PROCESS      MODULE
3
4 ! VERS. 1.8      !
5
6 CONSTANT
7 WRITE           := %2FC2      ! TYWR MON CALL !
8 WRITFLN        := %2FC2      ! PUTMSG MON CALL !
9 CRLF           := %2FD4      ! MCN CALL      !
10 RETURN_TO_MONITOR := %A902    ! HBUG REENTRY  !
11
12 COUNT := 10
13 TIME  := 500
14
15 SPACE := %20
16 DASH  := %2D
17
18 IO_MGR      := %20
19 FILE_MGR    := %40
20 MEM_MGR     := %46
21 CREATE_ENTRY_CODE := 50
22 INVALID_MMGR_CODE := 60
23 DELFT_ENTRY_CODE := 51
24 ACTIVATE_SEG_CODE := 52
25 DEACTIVATE_SEG_CODE := 53
26 SWAP_IN_SEG_CODE := 54
27 SWAP_OUT_SEG_CODE := 55
28 SUCCEEDED
29
30 IPAGE

```

```

31 TYPE
32 ADDRESS
33 SEG_DESC_REG RECORD [BASE_ADDR ADDRESS
34 LIMIT BYTE
35 ATTRIBUTE BYTE]
36 MMU RECORD [SDR_ARRAY [64 SEG_DESC_REG]]
37
38 !RLKS_USED WORD
39 MAX_FLKS WORD]
40 !NOTE: LAST TWO MMU COMPONENTS LEFT
41 OFF FOR CONVENIENCE SINCE ARE NOT
42 USED FOR THE SEG MGR DEMO!
43
44 PROCEDURE
45 SIGNAL
46 WAIT
47
48 GLOBAL
49 $SECTION MMU_DATA
50 MMU_IMAGE
51 $SECTION MM_DATA
52 G_AST_LOCK
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

0000

0000 0000

ARRAY [4 MMU]
WORD := 0

```

!
53 INTERNAL
54
55 I * * * MESSAGES * * * * I
56 IO ARRAY [* BYTE] := '%LE(FOR IO)'
57 FM ARRAY [* BYTE] := '%08(FOR FM)'
58 MM_MSG_1 ARRAY [* BYTE] := '%12KERNEL = SIGNALLER'
59 CREATE_MSG ARRAY [* BYTE] := '%10MM: CREATE_ENTRY'
60 DELETE_MSG ARRAY [* BYTE] := '%10MM: DELETE_ENTRY'
61 IPAGE

```

```

0002 08 28 46
0005 4F 52 20
0008 49 4F 29
000B 08 28 46
000E 4F 52 20
0011 46 4D 29
0014 12 4B 45
0017 52 4E 45
001A 4C 20 3D
001D 20 53 49
0020 47 4E 41
0023 4C 4C 45
0026 52
0027 13 4D 4D
002A 3A 20 43
002D 52 45 41
0030 54 45 5F
0033 45 4E 54
0036 52 59
0038 13 4D 4D
003F 3A 20 44
003E 4D 4C 45
0041 54 45 5F
0044 45 4E 54
0047 52 59

```



```

72 INTERNAL
73
74 $SECTION MM PROC
75
76 MM MAIN PROCEDURE
77 ENTRY
78 DO !** DO FOREVER **!
79   LDA RE,MM_MSG_ARRAY[0]
80   CALL WAIT
81   SENDER, R1 ISAVE SIGNALING PROC #!
82   R3,#50
83   MM_PRINT BLANKS
84   R2,MM_MSG_1
85   WRITELN
86   R1,SENDER
87   R1
88   CASE #IO MGR THEN LD R2,#IO
89
90   CALL WRITELN
91   CASE #FILE MGR THEN LD R2,#FM
92
93   CALL WRITELN
94
95   MM_DELAY
96   CRLF
97   R3,#50
98   MM_PRINT PLANKS
99   RH1,MM_MSG_ARRAY[0]
100  RL1,MM_MSG_ARRAY[1]
101
102  FI
103  CALL
104  CALL
105  LD
106  CALL
107  LDR
108  LDI
109  !PAGE

```

```

!
005C 0P01 0032
0060 5E0E 006C
0064 5F00 00E8
0068 5F0E 00C0
006C 0B01 0033
0070 5E0E 007C
0074 5F00 00F6
0078 5E08 00C0
007C 0B01 0034
0080 5F0F 00EC
0084 5F02 0104
0088 5E08 00C0
008C 0B01 0035
0090 5E0E 009C
0094 5F00 011A
0098 5F0E 00C0
009C 0B01 0036
00A0 5E0E 00AC
00A4 5F00 0128
00A8 5E08 00C2
00AC 0B01 0037
00B0 5E0F 00EC
00B4 5F00 0136
00B8 5E08 00C0
00BC 2102 007E

100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116 !PAGE

IF
R1
CASE #CREATE_ENTRY_CODE THEN
    CALL CREATE_ENTRY
CASE #DELETE_ENTRY_CODE THEN
    CALL DELETE_ENTRY
CASE #ACTIVATE_SEG_CODE THEN
    CALL ACTIVATE
CASE #DEACTIVATE_SEG_CODE THEN
    CALL DEACTIVATE
CASE #SWAP_IN_SEG_CODE THEN
    CALL SWAP_IN
CASE #SWAP_OUT_SEG_CODE THEN
    CALL SWAP_OUT
LD
R2,#ERROR_MSG
ELSE
FI

```

117	04CC 5F00	0FC0	CALL	WRITELN	
118	00C4 5F00	0144	CALL	MM_DELAY	
119	00C8 5F00	0FD4	CALL	CRIF	
120	04CC 2103	004F	LD	R3,#75	
121	00D0 5F02	0160	CALL	MM_PRINT_LINE	
122	00D4 5F00	0FD4	CALL	CRIF	
123			! ** SIGNAL (SENDER, 'DONE') ** !		
124	00DE 6101	00AC	LD	R1, SENDER	
125	00DC 760B	009B	LDA	R8,MM_MSG_ARRAY[0]	
126	00F0 5F00	0000*	CALL	SIGNAL	
127	00E4 E88D		OD ! ** REPEAT FOREVER ** !		
128	00E6 9E08		RET		
129	04FE		END MM_MAIN		
130					
131					
132			CREATE_ENTRY	PROCEDURE	
133					
134			ENTRY		
135	00FE 7608	009B	LDA	R8,MM_MSG_ARRAY[0]	
136	00EC 0C85	0202	LDE	GR8,#SUCCEEDED	
137	00F0 2102	0027	LD	R2,#CREATE_MSG	
138	04F4 9E0E		RET		
139	00F6		END CREATE_ENTRY		
140					
141					
142			DELETE_ENTRY	PROCEDURE	
143					
144			ENTRY		
145	00F6 7628	009B	LDA	R8,MM_MSG_ARRAY[0]	
146	00FA 0C85	0202	LDB	GR8,#SUCCEEDED	
147	04FF 2102	0038	LD	R2,#DELETE_MSG	
148	2102 9E0E		RET		
149	0104		END DELETE_ENTRY		
150					
151					! PAGE

!					
0104			ACTIVATE	PROCEDURE	
			ENTRY		
0104	7608	009B	LDA	R5,MM_MSG_ARRAY[0]	
010E	7609	00EB	LDA	R9,RET_VALUES[0]	
010C	2102	0010	LD	R2,#16	
0110	BA91	0280	LDIRB	QRB,QR9,R2	
0114	2102	0049	LD	R2,#ACTIVATE_MSG	
011E	9E08		RET		
011A			END ACTIVATE		
			DEACTIVATE	PROCEDURE	
011A			ENTRY		
011A	7608	009B	LDA	R5,MM_MSG_ARRAY[0]	
011E	0C85	0202	LDR	QRB,#SUCCEEDED	
0122	2102	0056	LD	R2,#DEACTIVATE_MSG	
0126	9E08		RET		
012E			END DEACTIVATE		
			SWAP_IN	PROCEDURE	
012E			ENTRY		
012E	760E	009B	LDA	R3,MM_MSG_ARRAY[0]	
012C	0C85	0202	LDF	GRE,#SUCCEEDED	
0130	2102	0065	LD	R2,#SWAP_IN_MSG	
0134	9E0E		RET		
0136			END SWAP_IN		

1 PAGE

185	!	0136	SWAP_OUT	PROCEDURE
186				
187			ENTRY	
188	0136	7608	LDA	R2,MM MSG_ARRAY[0]
189	013A	0CE5	LDR	QR8,#SUCCEEDED
190	013E	2102	LD	R2,#SWAP_OUT_MSG
191	0142	9E08	RET	
192	0144		END SWAP_OUT	
193				
194				
195				
196				
197	0144		MM_DELAY	PROCEDURE
198			!	*****!
199			!	PRODUCES 2 SEC DELAY !
200			!	*****!
201			ENTRY	
202	0144	2102	ID	R2, #COUNT
203	0148	2101	LD	R1, #TIME
204			DC	
205	014C	0B02	CP	R2, #0
206	0150	5E0E	IF EQ THEN	EXIT FI
207	0154	5E0E		
208	0156	AB20	DEC	R2
209	015A	7B1D	MREQ	R1
210	015C	FEF7	OD	
211	015E	9F28	RET	
212	0160		END MM_DELAY	
213				!PAGE


```

2 FM_PROCESS      MODULE
3
4 ! VERS 1.8      !
5 CONSTANT
6 WRITE           := %0FC8
7 Writeln        := %0FC0
8 CRLF           := %0FD4
9 RETURN_TO_MONITOR := %A902
10
11 COUNT := 10
12 TIME  := 500
13
14 SPACE := %20
15 DASH  := %2D
16
17 IO_MGR := %20
18 FILE_MGR := %40
19 MEM_MGR  := %00
20
21 EXTERNAL
22 SIGNAL
23 WAIT
24 CREATE_SEG
25 DELETE_SEG
26 SM_SWAP_IN
27 SM_SWAP_OUT
28 TERMINATE
29 MAKE_KNOWN
30
31 ! PAGE

```

```

PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE

```

!

```
32 $SECTION FM DATA
33 INTERNAL
34
35 ! * * * * MESSAGES * * * * !
36 FM_MSG_1 ARRAY [* BYTE] := '%12FM: IO = SIGNALLER'

37 FM_MSG_2 ARRAY [* BYTE] := '%17FM: CALL KERNEL(CREATE)'

38 FM_MSG_3 ARRAY [* BYTE] := '%16FM: RETURN FROM KERNEL'

39 FM_MSG_4 ARRAY [* BYTE] := '%17FM: CALL KERNEL(DELETE)'
```

```
0000 12 46 4D
0003 3A 20 49
0006 4F 20 3D
0009 20 53 49
000C 47 4E 41
000F 4C 4C 45
0012 52
0013 17 46 4D
0016 3A 20 43
0019 41 4C 4C
001C 2C 4B 45
001F 52 4E 45
0022 4C 28 43
0025 52 45 41
0028 54 45 29
002B 1C 46 4D
002F 3A 20 52
0031 45 54 55
0034 52 4E 20
0037 46 52 4F
003A 4D 20 4F
003D 45 52 4E
0040 45 4C
0042 17 46 4D
0045 3A 20 43
0048 41 4C 4C
004B 20 4B 45
004E 52 4E 45
0051 4C 28 44
0054 4D 4C 45
0057 54 45 29
```

40 !PAGE

```

!
005A 1E 46 4L 41 FM_MSG_5 ARRAY [* PYTE] := '%1EPM: CALL KERNEL(SWAP_IN)'
005D 3A 20 43
0060 41 4C 4C
0063 20 4B 45
0066 52 4F 45
0069 4C 28 53
006C 57 41 50
006F 5F 49 4E
0072 29
0073 19 46 4D 42 FM_MSG_6 ARRAY [* EYTE] := '%19PM: CALL KERNEL(SWAP_CUT)'
0076 3A 20 43
0079 41 4C 4C
007C 20 4F 45
007F 52 4E 45
0082 4C 28 53
0085 57 41 50
0088 5F 4F 55
008F 54 29
0090 3A 20 46 4D 43 FM_MSG_7 ARRAY [* PYTE] := '%1APM: CALL KERNEL(TERMINATE)'
0093 41 4C 4C
0096 20 4F 45
0099 52 4E 45
009C 4C 28 54
009F 45 52 4D
00A2 49 4E 41
00A5 54 45 29
00A8 1F 46 4D 44 FM_MSG_8 ARRAY [* PYTE] := '%1BPM: CALL KERNEL(MAKE_KNOWN)'
00AB 3A 20 43
00AE 41 4C 4C
00B1 20 4B 45
00B4 52 4E 45
00B7 4C 28 4D
00BA 41 4B 45
00BD 5F 4F 4E
00C0 4F 57
00C3 29

```

45 !PAGE

!	00C4	SENDER	WORD	
00C6		FM MSG ARRAY	ARRAY [16 BYTE]	
		INTERNAL		
		SECTION FM PROC		
0000		FM MAIN PROCEDURE		
		ENTRY		
		DO !** DO FOREVER **!		
		IFIRST MSG FROM IO PROC (CREATE)!		
0000	5F00	CALL WAIT_CALL		
0004	5F00	CALL PRINT_IO IS SIGNALLER		
0008	5F00	CALL CREATE_CALL		
000C	5F00	CALL FM_PRINT_RET FROM KER		
		ISIGNAL IO PROC FINISHED!		
0010	5F00	CALL FM_SIGNAL_CALL		
		MSG FROM IO PROC (MK KNOWN & SWAP_IN)!		
0014	5F00	CALL WAIT_CALL		
0018	5F00	CALL PRINT_IO IS SIGNALLER		
001C	5F00	CALL MK_CALL		
0020	5F00	CALL FM_PRINT_RET FROM KER		
0024	5F00	CALL SWAP_IN_CALL		
0028	5F00	CALL FM_PRINT_RET FROM KER		
		ISIGNAL IO PROC FINISHED!		
002C	5F00	CALL FM_SIGNAL_CALL		
		MSG FROM IO PROC (SWAP_OUT & TERMINATE)!		
0030	5F00	CALL WAIT_CALL		
0034	5F00	CALL PRINT_IO IS SIGNALLER		
0038	5F00	CALL SWAP_OUT_CALL		
003C	5F00	CALL FM_PRINT_RET FROM KER		
0040	5F00	CALL TERMINATE_CALL		
0044	5F00	CALL FM_PRINT_RET FROM KER		
		ISIGNAL IO PROC FINISHED!		
0048	5F00	CALL FM_SIGNAL_CALL		
		INEXT MSG FROM IO PROC (DELETE)!		
004C	5F00	CALL WAIT_CALL		
				80 !PAGE

!			
00450	5F00	017E'	81
0054	5F00	00AA'	82
0058	5F00	0138'	83
0250	5F00	0064'	84
0060	E3CF		85
0062	9F0E		86
0064			87
			88
			89
			90
			91
			92
			93
0064			94
			95
00C4	6101	00C4'	96
0068	760E	00C6'	97
006C	5F00	0000*	98
0070	9E08		99
0072			100
			101
			102
			103
			104
0072	760E	00C6'	105
0076	5F00	0000*	106
007A	6F01	00C4'	107
007F	9F0E		108
0080			109
			110
			111

CALL PRINT IO IS SIGNALLER
 CALL DELETE CALL
 CALL FM_PRINT_RET FROM KER
 ISIGNAL IO PROC FINISHED!
 CALL FM_SIGNAL_CALL
 OD !**REPEAT FOREVER**!
 RET
 END FM_MAIN

FM_SIGNAL_CALL PROCEDURE
 ENTRY
 LD R1, SENDER
 LDA R0, FM MSG ARRAY[0]
 CALL SIGNAL
 RET
 END FM_SIGNAL_CALL

WAIT_CALL PROCEDURE
 ENTRY
 LDA R0, FM MSG ARRAY[0]
 CALL WAIT
 LD SENDER, R1 ! SAVE SIGNALING PROCESS # !
 RET
 END WAIT_CALL

!	22E0	112	CREATE CALL	PROCEDURE
		113		
		114	ENTRY	
0050	2103 0019	115	LD	R3,#25
0084	5F00 0106	116	CALL	FM_PRINT PLANKS
00E6	2102 0013	117	LD	R2,#FM MSG 2
00EC	5F00 015E	118	CALL	FM_PRINT MSG
0090	2101 000A	119	LD	R1,#10 IMENTOR SEG #!
0094	2102 0001	120	LD	R2,#1 IENTRY #!
0098	2103 0030	121	LD	R3,#4E !SIZE!
009C	2104 0003	122	LD	R4,#3 IUPPER WORD OF CLASS!
02A0	2105 0001	123	LD	R5,#1 ILOWER WORD OF CLASS!
00A4	5F00 0000*	124	CALL	CREATE_SEG
0FAE	GE0E	125	RET	
02AA		126	END CREATE CALL	
		127		
		128		
02AA		129	DELETE CALL	PROCEDURE
		130		
		131	ENTRY	
00AA	2103 0019	132	LD	R3,#25
00AE	5F00 2106	133	CALL	FM_PRINT PLANKS
00F2	2102 0042	134	LD	R2,#FM MSG 4
00B6	5F00 015E	135	CALL	FM_PRINT MSG
00BA	2101 000A	136	LD	R1,#10 IMENTOR SEG #!
00FE	2102 0001	137	LD	R2,#1 IENTRY #!
00C2	5F00 0000*	138	CALL	DELETE_SEG
00C6	9E08	139	RET	
04CE		140	END DELETE CALL	
		141		
		142	!PAGE	

172	!	0104		SWAP_IN_CALL	PROCEDURE
173					
174				ENTRY	
175		0019		LD	R3,#25
176		0106		CALL	FM_PRINT BLANKS
177		005A		LD	R2, #FM MSG.5
178		015E		CALL	FM_PRINT MSG
179		000B		LD	R1,#11 ! SEG #!
180		0000*		CALL	SM_SWAP_IN
181				RET	
182				END SWAP_IN_CALL	
183					
184					
185					
186				SWAP_OUT_CALL	PROCEDURE
187					
188				ENTRY	
189		0019		LD	R3,#25
190		0106		CALL	FM_PRINT BLANKS
191		0073		LD	R2, #FM MSG.6
192		015E		CALL	FM_PRINT MSG
193		000B		LD	R1,#11 ! SEG #!
194		0000*		CALL	SM_SWAP_OUT
195				RET	
196				END SWAP_OUT_CALL	
197					
					!PAGE
211E					
011E		2103			
0122		5F00			
0126		2102			
012A		5F00			
012E		2101			
0132		5F00			
0136		9E08			
0138					

LINE	ADDRESS	DATA	INSTR	OPER	COMMENT	PROCEDURE
198						
199						
200	00138				FM_PRINT_RET_FROM_KER	PROCEDURE
201						
202					ENTRY	
203	00138	5F02	0192		CALL FM_DELAY	
204	0013C	2103	0019		LD R3,#25	
205	00140	5F00	0106		CALL FM_PRINT_BLANKS	
206	00144	2102	002B		LD R2,#FM_MSG 3	
207	00148	5F00	0FC0		CALL WRITELN	
208	0014C	5F00	0FD4		CALL CRLF	
209	00150	2103	004B		LD R3,#75	
210	00154	5F00	01AE		CALL FM_PRINT_LINE	
211	00158	5F00	0FD4		CALL CRLF	
212	0015C	9E08			RET	
213	0015E				END FM_PRINT_RET_FROM_KER	
214						
215						
216						
217	0015F				FM_PRINT_MSG	PROCEDURE
218						
219					ENTRY	
220	0015F	5F00	0FC0		CALL WRITELN	
221	00162	5F02	0192		CALL FM_DELAY	
222	00166	5F00	0FD4		CALL CRLF	
223	0016A	2103	004F		LD R3,#75	
224	0016E	5F00	01AE		CALL FM_PRINT_LINE	
225	00172	5F00	0FD4		CALL CRLF	
226	00176	9E08			RET	
227	00178				END FM_PRINT_MSG	
228						
229						
230						!PAGE

0178

0178 2103 0019
017C 5F00 0106
0180 2102 0000
0184 5F00 0FC0
0188 5F00 0FD4
018C 5F00 0192
0190 9E08
0192

0192

0192 2102 000A
0196 2101 01F4
019A 0F02 0000
019E 5E0E 01A6
01A2 5E08 01AC
01A6 AF20
01AE 7B1D
01AA E3F7
01AC 9E0E
01AE

PROCEDURE

PRINT IO IS SIGNALLER

ENTRY
LD R3,#25
CALL FM_PRINT BLANKS
LD R2,#FM MSG 1
CALL WRITELN
CALL CRLF
CALL FM_DELAY
RET
END PRINT IO IS SIGNALLER

FM_DELAY

PROCEDURE
!*****!
! PRODUCES 2 SEC DELAY !
!*****!

ENTRY

LD R2,#COUNT
LD R1,#TIME
DO
CP R2,#0
IF EQ THEN EXIT FI

DEC R2
MREQ R1
OD
RET
END FM_DELAY

!PAGE

!	01AE	263	FM_PRINT_LINE	PROCEDURE	!
		264		! *****	!
		265		! PRINTS LINE LENGTH	!
		266		! SPEC IN R3.	!
		267		! *****	!
		268	ENTRY		
	01AE C92D	269	LDR	RL0, #DASH	
		270	DO		
	01B0 0R03 0000	271	CP	R3, #0	
	01P4 5E0E 01BC	272	IF EQ THEN EXIT FI		
	01BE 5F0E 01C4				
	01PC 5F00 0FC8	273	CALL WRITE		
	01C0 4R30	274	DEC	R3	
	01C2 8EF6	275	OD		
	01C4 9R28	276	RET		
	01C6	277	END FM_PRINT_LINE		
		278			
		279			
	01C6	280	FM_PRINT_BLANKS	PROCEDURE	!
		281		! *****	!
		282		! PRINTS NUMBER OF	!
		283		! BLANKS SPEC IN R3.	!
		284		! *****	!
		285	ENTRY		
	01C6 C820	286	LDR	RL0, #SPACE	
		287	DO		
	01C8 0R03 0000	288	CP	R3, #0	
	01CC 5E0E 01D4	289	IF EQ THEN EXIT FI		
	01D0 5F0E 01DC				
	01D4 5F00 0FC8	290	CALL WRITE		
	01D8 4R30	291	DEC	R3	
	01DA 8EF6	292	OD		
	01DC 9E28	293	RET		
	01DE	294	END FM_PRINT_PLANKS		
		295			
		296	END FM_PROCESS		
		297	IPAGE		

```

2 IO_PROCESS      MODULE
3
4 ! VERS 1.6      !
5
6 CONSTANT
7 WRITE           := %0FCS      ! TYWR MON CALL !
8 Writeln         := %0FC0      ! PUTMSG MON CALL !
9 CRlf            := %0FD4      ! MON CALL      !
12 RETURN_TO_MONITOR := %A902    ! HBUG REENTRY  !
11
12 COUNT := 10
13 TIME  := 500
14
15 SPACE := %20
16 DASH  := %2D
17
18 IO_MGR := %2F
19 FILE_MGR := %40
20 MEM_MGR  := %00
21
22 EXTERNAL
23 MAKE_KNOWN
24 TERMINATE
25 SM_SWAP_IN
26 SM_SWAP_OUT
27 SIGNAL
28 WAIT
29 IPAGE

```

```

PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE
PROCEDURE

```



```

! 0056 19 49 4F 43 4C 45 4B 4D 4E 4F 50 51 52 53 54 55
0059 3A 20 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
005C 41 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
005F 20 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
0062 52 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B
0065 4C 28 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
0068 57 41 4F 4E 4D 4C 4B 4A 49 48 47 46 45 44 43
006B 5F 4F 4E 4D 4C 4B 4A 49 48 47 46 45 44 43 42
006E 54 29 4F 43 4C 4D 4E 4F 50 51 52 53 54 55 56 57
0070 1A 49 48 47 46 45 44 43 42 41 40 3F 3E 3D 3C
0073 3A 20 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
0076 41 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
0079 20 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
007C 52 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B
007F 4C 28 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
0082 45 52 4E 4F 50 51 52 53 54 55 56 57 58 59
0085 49 4E 4D 4C 4B 4A 49 48 47 46 45 44 43 42 41
0088 54 45 44 43 42 41 40 3F 3E 3D 3C 3B 3A 39 38
008F 16 49 48 47 46 45 44 43 42 41 40 3F 3E 3D 3C
0092 3A 20 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
0095 45 52 4E 4F 50 51 52 53 54 55 56 57 58 59
0098 46 52 4E 4F 50 51 52 53 54 55 56 57 58 59
009A 4D 20 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
009D 45 52 4E 4F 50 51 52 53 54 55 56 57 58 59
00A0 45 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59
00A2

```

```

38 IO_MSG_4 ARRAY [* BYTE] := '%19IO: CALL KERNEL(SWAP_OUT)'

39 IO_MSG_5 ARRAY [* BYTE] := '%1AIO: CALL KERNEL(TERMINATE)'

40 IO_MSG_6 ARRAY [* BYTE] := '%16IO: RETURN FROM KERNEL'

41 IO_MSG_ARRAY ARRAY [16 BYTE]
42 IPAGE

```

```

!
43 INTERNAL
44
45 $SECTION IO PROC
46
47
48 IO MAIN PROCEDURE
49 ENTRY
50 DO !** DO FOREVER **!
51 ! ** PRINT: "READ COMMAND#" ** !
52 LD R2, #IO_MSG_1
53 CALL IO_PRINT_MSG
54 ! ** SIGNAL FM TO CREATE ** !
55 ! ** AND WAIT ** !
56 CALL IO_SIGNAL_CALL
57 LDA R8, IO_MSG_ARRAY[0]
58 CALL WAIT
59 LD R2, #IO_MSG_1
60 CALL IO_PRINT_MSG
61 ! ** SIGNAL FM TO MAKE KNOWN & SWAP_IN ** !
62 ! ** AND WAIT ** !
63 CALL IO_SIGNAL_CALL
64 LDA R8, IO_MSG_ARRAY[0]
65
66 CALL WAIT
67 LD R2, #IO_MSG_1
68 CALL IO_PRINT_MSG
69 ! IC "READ" -- MAKE KNOWN AND SWAP_IN !
70 CALL IO_MK_CALL
71 CALL IC_SWAP_IN_CALL
72 LD R2, #IO_MSG_1
73 CALL IO_PRINT_MSG
74 ! SIGNAL FM TO SWAP OUT AND TERMINATE !
75 CALL IO_SIGNAL_CALL
76 LDA R8, IO_MSG_ARRAY[0]
77 ! PAGE

```

1	0048	5F00	0000*	78	CALL	WAIT
	004C	2102	0000'	79	LD	R2,#10 MSG 1
	0050	5F02	0052'	80	CALL	IO_PRINT MSG
	0054	5F00	00D6'	81	!	IO SWAP OUT AND TERMINATE !
	0056	5F00	00E8'	82	CALL	IO_SWAP_OUT CALL
	005C	2102	0000'	83	CALL	IO_TERMINATE_CALL
	0060	5F00	0082'	84	LD	R2,#10 MSG 1
	0064	5F00	0074'	85	CALL	IO_PRINT MSG
	0068	7608	02A2'	86	!	SIGNAL IM TO DELETE !
	006C	5F00	0000*	87	CALL	IO_SIGNAL_CALL
	0070	1EC7		88	LDA	R6,IO_MSG_ARRAY[0]
	0072	9E06		89	CALL	WAIT
	0074			90	OD	! **REPEAT FOREVER** !
				91	RET	
				92	END	IO_MAIN
				93		
				94		
	0074			95	IO_SIGNAL_CALL	PROCEDURE
				96		
				97	ENTRY	
	0074	2101	0040	98	LD	R1, #FILE_MGR !LD SIGNED PROC #!
	0076	760E	0011'	99	LDA	R6,IO_SEND[C]
	007C	5F00	0000*	100	CALL	SIGNAL
	0080	9E08		101	RET	
	0082			102	END	IO_SIGNAL_CALL
				103	!	PAGE

!	00E2	104	IO_PRINT_MSG	PROCEDURE
		105		
	00E2 5F00	106	ENTRY	
	00E6 5F00	107	CALL WRITELN	
	008A 5F00	108	CALL CRLF	
	00E2 2103	109	CALL IO_DELAY	
	0092 5F00	110	LD R3, #75	
	0096 5F00	111	CALL IO_PRINT_LINE	
	009A 9F0E	112	CALL CRLF	
	009C	113	RET	
		114	END IO_PRINT_MSG	
		115		
	009C	116	IO_PRINT_RET_FROM_KER	PROCEDURE
		117		
	009C 5F00	118	ENTRY	
	00A0 2102	119	CALL IO_DELAY	
	00A4 5F00	120	LD R2, #IO_MSG_6	
	00AE 9EE8	121	CALL IO_PRINT_MSG	
	00AA	122	RET	
		123	END IO_PRINT_RET_FROM_KER	
		124		
		125		
		126		
	00AA	127	IO_MK_CALL	PROCEDURE
		128		
		129	ENTRY	
	00AA 2102	130	LD R2, #IO_MSG_2	
	00AF 5F00	131	CALL IO_PRINT_MSG	
	00B2 2101	132	LD R1, #10 !MENTOR_SEG_NO!	
	00B6 2102	133	LD R2, #1 !ENTRY_NO!	
	00BA 2103	134	LD R3, #1 !RYAD_ACCESS_DESIRE!	
	00BE 5F00	135	CALL MAKE_KNOWN	
	00C2 9E38	136	RET	
	00C4	137	END IO_MK_CALL	
		138		!PAGE

!	00C4		IO_SWAP_IN_CALL	PROCEDURE
			ENTRY	
00C4	2102	003D	LD	R2,#IO MSG 3
00CE	5F00	0082	CALL	IO_PRINT MSG
00CC	2101	000B	LD	R1,#11 ISEG #1
00D0	5F00	0000*	CALL	SM_SWAP_IN
00D4	9E0E		RET	
00D6			END IO_SWAP_IN_CALL	
00D6			IO_SWAP_OUT_CALL	PROCEDURE
			ENTRY	
00D6	2102	0056	LD	R2,#IO MSG 4
00DA	5F00	0082	CALL	IO_PRINT MSG
00DF	2101	000B	LD	R1,#11 ISEG #1
00E2	5F00	0000*	CALL	SM_SWAP_OUT
00E6	9E08		RET	
00E8			END IO_SWAP_OUT_CALL	
00E8			IO_TERMINATE_CALL	PROCEDURE
			ENTRY	
00E8	2102	0070	LD	R2,#IO MSG 5
00EC	5F00	0082	CALL	IO_PRINT MSG
00F0	2101	000B	LD	R1,#11 ISEG #1
00F4	5F00	0000*	CALL	TERMINATE
00F8	9E08		RET	
00FA			END IO_TERMINATE_CALL	

170 IPAGE

LIST OF REFERENCES

1. Coleman, A. R., Security Kernel Design for a Microprocessor-Based, Multilevel, Archival Storage System, MS Thesis, Naval Postgraduate School, December 1979.
2. Conway, T., and others, Introduction to Microprocessors Programming Using PLZ, Winthrop Publishers, 1979.
3. Denning, D.E., "A Lattice Model of Secure Information Flow," Communications of the ACM, v. 19 p. 236-242, May 1976.
4. Gary, A.V. and Moore, E.E., The Design and Implementation of the Memory Manager for a Secure Archival Storage System, MS Thesis, Naval Postgraduate School, June 1980.
5. Madnick, S. E., and Donovan, J.J., Operating Systems, McGraw Hill, 1974.
6. O'Connell, J. S., and Richardson, L. D., Distributed Secure Design for a Multi-microprocessor Operating System, MS Thesis, Naval Postgraduate School, June 1979.
7. Parks, E. J., The Design of a Secure File Storage System, MS Thesis, Naval Postgraduate School, December, 1979.
8. Reed, P. D., and Kanodia, R. K., "Synchronization With Eventcounts and Sequencers," Communications of the ACM, v. 22 no. 2 p. 115-124, February 1979.
9. Reitz, S. L., An Implementation of Multiprogramming and Process Management for a Security Kernel Operating System, MS Thesis, Naval Postgraduate School, June 1980.
10. Schell, Lt.Col. R. R., "Computer Security: the Achilles Heel of the Electronic Air Force?," Air University Review, v. 30 no. 2 p. 16-33, January 1979.
11. Schell, Lt.Col. R. R., "Security Kernels: A Methodical Design of System Security," USE Technical Papers (Spring Conference, 1979). pp 245-256, March 1979.

12. Snook, T., and others, Report on the Programming Language PLZ/SYS, Springer-Verlag, 1978.
13. Zilog, Inc., Z8000 PLZ/ASM Assembly Language Programming Manual, 03-3055-01, Revision A, April 1979.
14. Zilog, Inc., Z8001 CPU Z8002 CPU, Preliminary Product Specification, March 1979.
15. Zilog, Inc., Z8010 MMU Memory Management Unit, Preliminary Product Specification, October 1979.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4. Lyle A. Cox, Jr., Code 52C1 Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
5. LTCOL Roger R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
6. Joel Trimble, Code 221 Office of Naval Research 800 North Quincy Arlington, Virginia 22217	1
7. LCDR John T. Wells P.O. Box 366 Waynesboro, Mississippi 39367	2
8. Office of Research Administration Code 012A Naval Postgraduate School Monterey, California 93940	1
9. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
10. I. Larry Avrunin, Code 18 DTNSRDC Bethesda, Maryland 20084	1

- | | | |
|-----|--|---|
| 11. | R. P. Crabb, Code 9134
Naval Oceans Systems Center
San Diego, California 92152 | 1 |
| 12. | Kathryn Heninger, Code 7503
Naval Research Lab
Washington, D.C. 20375 | 1 |
| 13. | Dr. J. McGraw
U.C. - L.L.L. (1-794)
P.O. Box 808
Livermore, California 94550 | 1 |
| 14 | Mark Underwood
NPRDC
San Diego, California 92152 | 1 |
| 15. | Walter P. Warner, Code K70
NSWC
Dahlgren, Virginia 22448 | 1 |
| 16. | M. George Michael
U.C. - L.L.L. (L-76)
P.O. Box 808
Livermore, California 94550 | 1 |

DATE
ILME