IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A SECURE ARCHIVAL STORAGE

UNCLASSIFIED

1 of 3

09/25/80
IMPLEMENTATION OF SEGMENT MANAGEMENT
FOR A SECURE ARCHIVAL STORAGE SYSTEM

by

John Timothy Wells

September 1980

Thesis Advisor: R. R. Schell

proved for public release; distribution unlimited
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 microprocessor.
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. Stuckey
Thesis Advisor

Earl A. Cade
Second Reader

Chairman, Department of Computer Science

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

- 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

- 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. DFR 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
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.................................66
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.
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.


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.
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 Z8001 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 (NES) 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 \texttt{CLASS\_EQ} and \texttt{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.
Figure 2. Active Process Table
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.
**Figure 3. Virtual Processor Table**
VPT RECORD
  [LOCK WORD
   RUNNING_LIST VP_INDEX
   READY_LIST VP_INDEX
   FREE_LIST MSG_INDEX
   VP ARRAY
     [NF_VF VP_TABLE]
   MSG_O ARRAY
     [NR_VF MSG_TABLE]

VP_TABLE RECORD
  [DEF ADDRESS
   PPI WORD
   STATE WORD
   IDLE_FLAG WORD
   PREEMPT WORD
   PHYS_PROCESSOR WORD
   NEXT_READY VP_INDEX
   MSG_LIST 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, Z801 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.
<table>
<thead>
<tr>
<th>MODULE</th>
<th>INSTRUCTION SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Manager</td>
<td>Create_Segment</td>
</tr>
<tr>
<td></td>
<td>Delete_Segment</td>
</tr>
<tr>
<td></td>
<td>Make_Known</td>
</tr>
<tr>
<td></td>
<td>Terminate</td>
</tr>
<tr>
<td></td>
<td>SM_Swap_In</td>
</tr>
<tr>
<td></td>
<td>SM_Swap_Out</td>
</tr>
<tr>
<td>Event Manager</td>
<td>Read</td>
</tr>
<tr>
<td></td>
<td>Ticket</td>
</tr>
<tr>
<td>Non-Discretionary Security</td>
<td>Class_EC</td>
</tr>
<tr>
<td>Traffic Controller</td>
<td>Class_GF</td>
</tr>
<tr>
<td></td>
<td>Advance</td>
</tr>
<tr>
<td></td>
<td>Await</td>
</tr>
<tr>
<td></td>
<td>Process_Class</td>
</tr>
<tr>
<td>Inner Traffic Controller</td>
<td>Signal</td>
</tr>
<tr>
<td></td>
<td>Wait</td>
</tr>
<tr>
<td></td>
<td>Swap_VDBR</td>
</tr>
<tr>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td></td>
<td>Set_Preempt</td>
</tr>
<tr>
<td></td>
<td>Test_Preempt</td>
</tr>
<tr>
<td></td>
<td>Running_VP</td>
</tr>
<tr>
<td>Memory Manager</td>
<td>MM_Create_Entry</td>
</tr>
<tr>
<td></td>
<td>MM_Delete_Entry</td>
</tr>
<tr>
<td></td>
<td>MM_Activate</td>
</tr>
<tr>
<td></td>
<td>MM_Deactivate</td>
</tr>
<tr>
<td></td>
<td>MM_Swap_In</td>
</tr>
<tr>
<td></td>
<td>MM_Swap_Out</td>
</tr>
</tbody>
</table>

Figure 4. Extended Instruction Sets
<table>
<thead>
<tr>
<th>MODULE</th>
<th>DATABASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gate Keeper</td>
<td>Parameter Table</td>
</tr>
<tr>
<td>Segment Manager</td>
<td>Known_Segment_Table (KST)</td>
</tr>
<tr>
<td>Traffic Controller</td>
<td>Active_Process_Table (AFT)</td>
</tr>
<tr>
<td>Inner Traffic Controller</td>
<td>Virtual_Process_Table (VPT)</td>
</tr>
<tr>
<td>Memory Manager</td>
<td>Global_Active_Segment_Table (G_AST)</td>
</tr>
<tr>
<td></td>
<td>Local_active_Segment_Table (I_AST)</td>
</tr>
<tr>
<td></td>
<td>Memory_Management_Unit_Image (MM_Image)</td>
</tr>
<tr>
<td></td>
<td>Alias_Table</td>
</tr>
<tr>
<td></td>
<td>Disk_Bit_Map</td>
</tr>
<tr>
<td></td>
<td>Global_Memory_Bit_Map</td>
</tr>
<tr>
<td></td>
<td>Local_Memory_Bit_Map</td>
</tr>
</tbody>
</table>

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 embarassment 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 "w-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 hierarchically organized in a tree structure. The structure consists of nodes, leaves, and a root from which the tree eminates. 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 | 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

44
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 - LS = 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 \| oac)

(1) General - LS <,>, or = LO and \{CS\} \| \{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,{%} ) | (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, {%})

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/SYS 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_SegNr field is a number representing the segment number of a segment's parent or "mentor" segment. Its importance will discussed shortly.

The EntryNr 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_SegNr with the segment's EntryNr (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.
### Figure 6. Known Segment Table.

<table>
<thead>
<tr>
<th>Segment_#</th>
<th>MM_Handle</th>
<th>Size</th>
<th>Access_Mode</th>
<th>In_Core</th>
<th>Class</th>
<th>M_Seg_No</th>
<th>Entry_Number</th>
</tr>
</thead>
</table>

**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_Intege
- Entry_Number: Short_Integer

53
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**, **SWAP_OUT**.
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 Elk_S_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.
<table>
<thead>
<tr>
<th>Segment No.</th>
<th>Blocks Used</th>
<th>Max Avail Blocks</th>
<th>Phase Addr</th>
<th>Limit</th>
<th>Attributes</th>
</tr>
</thead>
</table>

**Figure 7. Memory Management Unit Image**

**Figure 6. 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 8000 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 MSC_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 to occur as previously discussed.

There are three basic cases that may occur in MAKEKNOWN: (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 NULLSEG 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 TERMINATE. 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 \texttt{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 \texttt{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 \texttt{MM\_SWAP\_IN}. It performs normal interface (IPC) functions and returns a success code from the memory manager process' \texttt{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 \texttt{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 \texttt{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 \texttt{SM\_SWAP\_OUT} is exactly analogous to that of \texttt{SM\_SWAP\_IN} except that the segment's KST \texttt{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 >= 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 | N | Nu | % 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 OR N

\[0000 \ 0000 \ 0000 \ 0001\] OR \[0000 \ 0000 \ 0000 \ 0010\]

= \[0000 \ 0000 \ 0000 \ 1011\] = \{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/STS 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_activActIVATE, 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/STS 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 CEMSGPTR (which points to a structure of type CREATE_MSG) set equal to a pointer (COMMSGPTR) to the actual message array (COMMSGBUF). 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_KNOW 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's 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_VDER (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 0000 | 0000 0040 | CCCC CCCC |          |          |
| 9210 | 0300 0003 | 0001 0000 | 0001 0020 | CCCC CCCC | (FM)     |          |
| 9220 | 0000 0000 | 0000 CCCC | CCCC CCCC | CCCC CCCC |          |          |
| 9230 | 0200 0003 | 0001 0000 | 0001 FFFF | CCCC CCCC | (IO)     |          |
| 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 |          |          |

Figure 9. Initialized Active Process Table
<table>
<thead>
<tr>
<th>D 9300 18 (FM, partial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000 000F 000F 0050 0100 0003 0001 0000</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 FF00</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 FF00</td>
</tr>
<tr>
<td><em>........P............</em></td>
</tr>
<tr>
<td><em>................</em></td>
</tr>
<tr>
<td>D 9700 18 (IO, partial)</td>
</tr>
<tr>
<td>0000 0000 000F 000F 0050 0100 0003 0001 0000</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 FF00</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 FF00</td>
</tr>
<tr>
<td><em>........P............</em></td>
</tr>
<tr>
<td><em>................</em></td>
</tr>
</tbody>
</table>

Figure 11. Initialized Known Segment Tables
<table>
<thead>
<tr>
<th>P 8000 10 (MM, partial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000 cccc cccc 7000 cccc 6000 cccc cccc cccc</td>
</tr>
<tr>
<td>8010 cccc cccc cccc cccc cccc cccc cccc cccc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P 8100 10 (IDLE, partial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8100 cccc cccc 7100 cccc 6100 cccc cccc cccc</td>
</tr>
<tr>
<td>8110 cccc cccc cccc cccc cccc cccc cccc cccc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P 8200 10 (IO, partial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8200 cccc cccc 7200 cccc 9700 cccc cccc cccc</td>
</tr>
<tr>
<td>8210 cccc cccc cccc cccc cccc cccc cccc cccc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P 8300 10 (PM, partial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8300 cccc cccc 7300 cccc 9300 cccc cccc cccc</td>
</tr>
<tr>
<td>8310 cccc cccc cccc cccc cccc cccc cccc cccc</td>
</tr>
</tbody>
</table>

Figure 12. Initialized Memory Management Unit Image
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
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

"M: 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
Figure 16. Generated Output (continued)
IO: RETURN FROM KERNEL

IO: READ COMMAND

FM: IO = SIGNALLER
FM: CALL KERNEL(SNAP OUT)

KERNEL = SIGNALLER(FOR FM)
MM: SWAP OUT

FM: RETURN FROM KERNEL

FM: CALL KERNEL(TERMINATE)

KERNEL = SIGNALLER(FOR FM)
MM: DEACTIVATE

"M: RETURN FROM KERNEL

IO: READ COMMAND

NMI

Figure 16. Generated Output (continued)
IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A SECURE ARCHIVAL STORAGE

SEP 80 J T WELLS

UNCLASSIFIED
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)

Figure 16. Generated Output (continued)
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 Z8022 sixteen bit nonsegmented microprocessor. Segmentation hardware (Zilog's Z8018 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 PL7/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_KSTYPES := ??!TO BE DETERMINED!
FALSE := 0
TRUE := 1
READ := 1
WRITE := 2

! ***** SUCCESS_CODES ***** !
SUCCEEDED := 2
SEGMENT NOT KNOWN := 22
ACCESS_CLASS NOT_EQ := 33
NOT_COMPATIBLE := 24
SEGMENT_TOO_LARGE := 25
NO_SEG_AVAL := 27
SEGMENT_NOT_KNOWN := 22
SEGMENT_IN_CORE := 29
KERNELSEGMENT := 33
INVALID_SEGMENT_NC := 31
NO_ACCESS_PERMITTED := 32
LEAFSEG_EXISTS := 13
NC_LEAF_EXISTS := 11
ALIAS_DOES_NOT_EXIST := 23
NO_CHILD_TO_DELETE := 22
LAST_FULL := 12
LAST_FULL := 13
LOCAL_MEMORY_FULL := 16
GLOBAL_MEMORY_FULL := 17
SEC_STORE_FULL := 21
PROC_CLASS_NOT_JE_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_NC SECRET_INTEGER
ENTRY_NUMBER SHORT_INTEGER ]

KST ARRAY [ MAX NO KST_ENTRIES KST_REC ]
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 LCNG 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 (PPCC_CLASS LCNG)

ITC_GET_SEG_PTR_PROCEDURE
RETURNS (SEGPTR SEG_ARRAY)

MONITOR_PROCEDURE
!TO BE IMPLEMENTED AT ASSEMBLY LEVEL - SIMPLY WILL CALL THE MONITOR AT ADDRESS $359A!

INTERNAL

HPTR ~H_ARRAY
KPTR ~ESTPTR
GLOBAL

CREATE_SEGMENT_PROCEDURE (MENTOR_SEG_NO SHORT_INT, ENTRY_NO SHORT_INT, CLASS LONG, SIZE VAL, *CRTL)

RETURNS (SUCCESS_CODE BYTE)

!* **NOTE: REENTRANT PROCEDURE ****!
! SAVE LOCAL VARIABLES ON STAGE TO ENSURE REENTRANT!
LOCAL M_SEG_INDEX SHORT_INT, ENTRY SHORT_INT

IF SIZE > MAX_SEG_SIZE THEN
  SUCCESS_CODE := SEGMENT_TOO_LARGE
ELSE
  M_SEG_INDEX := MENTOR_SEG_NO - NR_OF_SEGs
  KPTR := EST_PTR 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_PTR_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_EQ (CLASS, KPTR [M_SEG_INDEX].CLASS)
      IF CONDITION_CODE = FALSE THEN
        SUCCESS_CODE := NOT_COMPATIBLE
      ELSE
        KPTR := #KPTR [M_SEG_INDEX].MM_HANDLE
        SUCCESS_CODE := MM_CREATE_ENTRY (KPTR, ENTRY_NO, SIZE, CLASS)
        CONFINEMENT_CHECK (SUCCESS_CODE, FI
      FI
  FI
RETURN
END CREATE_SEGMENT
DELETE_SEGMENT_PROCEDURE. INVOKED BY SUPERVISOR
PROCEDURE VIA THE 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 SEGMENT
 ENTRY_NO SHORT_INTEGER )
RETURNS ( SUCCESS_CODE BYTE )

! **** ACTE: 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 ITG GET_SEGMENT PTR \ KSEG_NUM,
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_CE2CA ( SUCCESS_CODE )
FI
FI
RETURN
END DELETE_SEGMENT

123
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 := KSEPTR ITC_GET_SEC_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
AND IF 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

164
ELSE
  IF KPTR^[INDEX].M_SEG_NC = NULL_SEG
    AND IF AVAIL_SEG = NULL_SEG THEN
      AVAIL_SEG := INDEX + NR_OF_KSEG
      FI
    INDEX := 1
    IF INDEX > MAX_NC_KST_ENTRIES THEN EXIT FI
    FI
  OD
! EXIT IF KNOWN !
IF SEGMENT_NO = NULL_SEG
  AND IF AVAIL_SEG <> NULL_SEG THEN !CASE: SEGMENT NOT KNOWN AND SEG # IS AVAILABLE!
    INDEX := AVAIL_SEG - NR_OF_KSEG
    SEGMENT_NO := AVAIL_SEG
    LIR_NC := "TC.GETLIR NC"
    SUCCESS_CODE := KPTR^[INDEX].CLASS, KPTR^[INDEX].SIZE :=
      MM_ACTIVATE(DIR_NO, KPTR, ENTRY_NO, SEGMENT_NO),
    CONFINEMENT_CHECK (SUCCESS_CODE),
    IF SUCCESS_CODE = SUCCEEDED THEN
      CONDITION_CODE := CLASS_SEG (PROC_CLASS,
        KPTR^[INDEX].CLASS),
      IF CONDITION_CODE = FALSE THEN !NO ACCESS!
        ACCESS_ALLOWED := NULL_ACCESS
      SUCCESS_CODE := MM_DEACTIVATE (LIR_NC,E PTR),
      CONFINEMENT_CHECK (SUCCESS_CODE),
      SUCCESS_CODE := PROC_CLASS_NOT_RESSEG_CLASS
      SEGMENT_NC := NULL_SEG
    ELSE
      CONDITION_CODE := CLASS_SEG (PROC_CLASS,
        KPTR^[INDEX].CLASS),
      IF CONDITION_CODE = TRUE
        AND IF ACCESS_DESIREd = WRITE THEN
          ACCESS_ALLOWED := WRITE
        ELSE
          ACCESS_ALLOWED := READ
        FI
      KPTR^[INDEX].IN_CORE := FALSE
      KPTR^[INDEX].M_SEG_NO := MENTOP_SEG NO
      KPTR^[INDEX].ENTRY_NUMBER := ENTRY NC
      KPTR^[INDEX].ACCESS_MODE := ACCESS_ALLOWED
      FI
    ELSE
      SEGMENT_NO := NULL_SEG
      ACCESS_ALLOWED := NULL_ACCESS
    FI
  ELSE
    SUCCESS_CODE := NO_SEG_AVAIL
    SEGMENT_NC := NULL_SEG
    ACCESS_ALLOWED := NULL_ACCESS
  FI
  IF RETURN
     END MAKEKNOWN
**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_DEACTIVATE IS CALLED FOR MEMORY MANAGER ACTION.**

**TERMINATE PROCEDURE ( SEGMENT_NC SEGAT_INTEGER )**

RETURNS ( SUCCESS_CODE BYTE )

! **** NOTE: REENTRANT PROCEDURE **** !

! SAVE LOCAL VARIABLES ON STACK TO ENSURE REENTRANT !

LOCAL INDEX SHORT INTEGER

DER_NC SEGAT_INTEGER

ENTRY

INDEX := SEGMENT_NO - NR_OF_KSEGS
KPTR := KSPIPTR ITC_GET_SIG_PTR KS4 SEG_NC.
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_NC THEN
SUCCESS_CODE := INVALID_SEGMENT_NO
ELSE
KPTR := #KPTR INDEX].MM_HANDLE
DBR_NO := ITC_GET_DBR_NO
SUCCESS_CODE := MM_DEACTIVATE (DER_NC, EPI)
CONFIRMATION_CHECK ( SUCCESS_CODE )
IF SUCCESS_CODE = SUCCESSED THEN
KPTR INDEX].M_SEG_NO := NULL
FI
FI
FI
RETURN
END TERMINATE
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 - NT_OF_KSEGS
KPTR := KSTPTR ITG_GET_SEG_PTR ( KST_SEG_NO )
IF KPTR^INDEX] .SEG_NO = NULL_SEG THEN
SUCCESS_CODE := SEGMENT_NOT_KNOWN
ELSE
IF KPTR^ [INDEX] .IN.Core = TRUE THEN
SUCCESS_CODE := SUCCEEDED
ELSE
KPTR := #KPTR^ [INDEX] .MEM_BASE
DBR_NO := TC_GET_DBK_NO
SUCCESS_CODE := MM_SWAP_IN ( KPTR, DBR_NO, KPTR^ [INDEX].ACCESS_MODE,
CONFINEMENT_CHECK ( SUCCESS_CODE )
IF SUCCESS_CODE = SUCCEEDED THEN
KPTR^ [INDEX].IN.Core := TRUE
FI
FI
RETURN
END SM_SWAP_IN
**** SM_SWAP_OUT PROEDURE. INVOKED BY SUPERVISER. ****
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 PROEDURE ( 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
  EPTR := KSTPTR ITC_GET_SEG_PTR ( KST_SEG_NO )
        IF &EPT^ [INDEX] , M_SEG_NO = NULL SEG THEN
            SUCCESS_CODE := SEGMENT_NOT_KNOWN
        ELSE 
            IF &EPT^ [INDEX] , IN_Core = FALSE THEN
                SUCCESS_CODE := SUCCEEDED
            ELSE
                EPTR := &EPT^ [INDEX] , MM_HANDLE
                DBR_NO := TC_GET_DER_NO
                SUCCESS_CODE := MM_SWAP_OUT ( DBR_NO, EPTR )
                CONFINEMENT CHECK ( SUCCESS_CODE )
                IF SUCCESS_CODE = SUCCEEDED THEN
                    &EPT^ [INDEX] , IN_Core := FALSE
                FI
            FI
        FI
RETURN
END SM_SWAP_OUT
CONFINEMENT_CHECK PROCEDURE, SERVICE PROCEDURE 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 %ZEGA.

CONFINEMENT_CHECK PROCEDURE (SUCCESS_CODE EXIT).

** 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_STOP FULL THEN
  MONITOR !EXIT SYSTEM!
F1
RETURN
END CONFINEMENT_CHECK
END SEGMENT_MANAGER
APPENDIX B - SEGMENT MANAGER PL/7/ASM LISTINGS

SEG_MGR MODULE

<table>
<thead>
<tr>
<th>CONSTANT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL_SEG</td>
<td>:= -1</td>
</tr>
<tr>
<td>NULL_ACCESS</td>
<td>:= 4</td>
</tr>
<tr>
<td>MAX_SEG_NG</td>
<td>:= 64</td>
</tr>
<tr>
<td>MAX_NO_KST_ENTRIES</td>
<td>:= 5+</td>
</tr>
<tr>
<td>MAX_SEG_SIZE</td>
<td>:= 128</td>
</tr>
<tr>
<td>KST_SFG_NO</td>
<td>:= 2</td>
</tr>
<tr>
<td>NF_OF_KSRGS</td>
<td>:= 10</td>
</tr>
<tr>
<td>TRUE</td>
<td>:= 1</td>
</tr>
<tr>
<td>FALSE</td>
<td>:= 0</td>
</tr>
<tr>
<td>READ</td>
<td>:= 1</td>
</tr>
<tr>
<td>WRITE</td>
<td>:= 3</td>
</tr>
</tbody>
</table>

*** SUCCESS Codes ***

<table>
<thead>
<tr>
<th>SUCCESS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESSDEL</td>
<td>:= 2</td>
</tr>
<tr>
<td>MENTCR_SEG_NOT_KNOWN</td>
<td>:= 22</td>
</tr>
<tr>
<td>ACCESS_CLASS_NOT_EQ</td>
<td>:= 23</td>
</tr>
<tr>
<td>NOT_COMPATIBLE</td>
<td>:= 24</td>
</tr>
<tr>
<td>SEGMENT_TOO_LARGE</td>
<td>:= 25</td>
</tr>
<tr>
<td>KSEG_SFG_AVAIL</td>
<td>:= 27</td>
</tr>
<tr>
<td>SEGMENT_NOT_KNOWN</td>
<td>:= 28</td>
</tr>
<tr>
<td>SEGMENT_IN_CORE</td>
<td>:= 29</td>
</tr>
<tr>
<td>KERNFL_SEGMENT</td>
<td>:= 30</td>
</tr>
<tr>
<td>INVALID_SEGMENT_NO</td>
<td>:= 31</td>
</tr>
<tr>
<td>NO_ACCESS_PERMITTED</td>
<td>:= 32</td>
</tr>
<tr>
<td>LEAF_SFG_EXISTS</td>
<td>:= 18</td>
</tr>
<tr>
<td>NO_LEAF_EXISTS</td>
<td>:= 11</td>
</tr>
<tr>
<td>ALIAS_DOES_NOT_EXIST</td>
<td>:= 23</td>
</tr>
<tr>
<td>NO_CTRL_TO_LEFTE</td>
<td>:= 24</td>
</tr>
<tr>
<td>G_AST_FUL</td>
<td>:= 12</td>
</tr>
<tr>
<td>LAST_FUL</td>
<td>:= 13</td>
</tr>
<tr>
<td>PROC_CLASS_NOT_OF_SFG_CLASS</td>
<td>:= 41</td>
</tr>
</tbody>
</table>
LOCAL MEMORY FULL := 16
GLOBAL MEMORY FULL := 17
SEC STOR FULL := 21
MONITOR := 4059A

TYPE L_ARRAY ARRAY [ 3 WORD ]

KST_REC RECORD
[ MM. HANDLE L<Array SIZE WORD
ACCESS MODE BYTE
IN. CORP BYTE
CLASS LONG
M_SEG. NO SHORT INTEGER
ENTRY. NUMBER SHORT INTEGER]

ADDRESS WORD

SEG. ARRAY ARRAY [ MAX. SEG_SIZE BYTE ]

INTERNAL

$SECTION KST LCL
NOTE: THIS SECTION IS AN OVERLAY/FRA CRM USED TO
DEFINE THE KST FORMAT. NO STORAGE IS ASSIGNED
RATHER THE KST IS STORED IN A SEPARATE
SEGMENT SET ASIDE FOR IT !

$ANS E
KST ARRAY [ MAX. NC. KST. ENTRIFS KST. REC ]

PAGE
EXTERNAL

CLASS_EQ PROCEDURE
CLASS_GE PROCEDURE
MM_CREATE_ENTRY PROCEDURE
MM_DELETE_ENTRY PROCEDURE
MM_ACTIVATE PROCEDURE
MM_DEACTIVATE PROCEDURE
MM_SWAP_IN PROCEDURE
MM_SWAP_OUT PROCEDURE
TC_GET_PROC_CLASS PROCEDURE
ITC_GET_SEG_PTR PROCEDURE
TC_GET_DBR NO PROCEDURE

IPAGE
PROCEDURE

***CHECKS VALIDITY OF CREATE REQUEST AND***
***CALLS MM CREATE IF VALID.***

***REGISTER USE:***

***PARAMETERS***

**R1:** MENTOR_SEG_NO (INPUT)

**R2:** ENTRY_NO (INPUT)

**R3:** SIZE (INPUT)

**R4:** CLASS (INPUT)

**R5:** SUCCESS_CODE (RETURN)

***LOCAL USE***

**R9:** KST_REC_INDEX

**R6, R7:** VARIOUS USES

**R13:** KST

***END***

ENTRY

0030 2433 0380
0164 5687 6016*
0065 2182 0019
0036 5838 63A2*
0416 03EF 666A
3014 1CF9 2104
0018 2131 0032
041C 5F8C 669V*
3323 A125
2072 1CF1 61C4

143 IPAGE
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>134</td>
<td>LD</td>
<td>R9,R1 ICOPY OF MENTOR_SEG_NO</td>
</tr>
<tr>
<td>135</td>
<td>SUB</td>
<td>R9,#NR_CF;ESGS ICOUNT MENTOR_SEG_NO KST_REC_INDEX</td>
</tr>
<tr>
<td>136</td>
<td>MULT</td>
<td>RR8,#SIZEOF KST_REC !OFFSET TO KST_REC</td>
</tr>
<tr>
<td>137</td>
<td>ADD</td>
<td>R13,R9 !ALD OFFSET TO KST_PASF ADDRESS</td>
</tr>
<tr>
<td>138</td>
<td>LD</td>
<td>RF6,#NULL_SEG</td>
</tr>
<tr>
<td>139</td>
<td>CPB</td>
<td>RL6,#ST_M_SEG_NO(R13)</td>
</tr>
<tr>
<td>140</td>
<td>IF_EQ</td>
<td>THEN !MENTOR_SEG NOTKNOWN</td>
</tr>
<tr>
<td>141</td>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>PUSH</td>
<td>@R15,R13</td>
</tr>
<tr>
<td>143</td>
<td>CALL</td>
<td>TC_GET_PROC_CLASS !(RR2:PROC_CLASS)!</td>
</tr>
<tr>
<td>144</td>
<td>FCN</td>
<td>R13,G15</td>
</tr>
<tr>
<td>145</td>
<td>LDL</td>
<td>RR4,KST_CLASS(R13)</td>
</tr>
<tr>
<td>146</td>
<td>PUSH</td>
<td>@R15,R13</td>
</tr>
<tr>
<td>147</td>
<td>CALL</td>
<td>CLASS_EQ !(RR2:PROC_CLASS)!</td>
</tr>
<tr>
<td>148</td>
<td></td>
<td>!(RR4:MENTOR_SEG_CLASS)!</td>
</tr>
<tr>
<td>149</td>
<td></td>
<td>!(R1:(RET)CONDITION_CODE)!</td>
</tr>
<tr>
<td>150</td>
<td>FCN</td>
<td>R13,G15</td>
</tr>
<tr>
<td>151</td>
<td>LDL</td>
<td>RF6,R1</td>
</tr>
<tr>
<td>152</td>
<td>LDM</td>
<td>R1,@R15,#5 !RESTORE INPUT REGS</td>
</tr>
<tr>
<td>153</td>
<td>CP</td>
<td>RF6,#FALSE</td>
</tr>
<tr>
<td>154</td>
<td>IF_EQ</td>
<td>THEN !</td>
</tr>
<tr>
<td>155</td>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>156</td>
<td>LD</td>
<td>RF,#ACCESS_CLASS_NOT_EQ</td>
</tr>
<tr>
<td>157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>158</td>
<td>PUSH</td>
<td>@R15,R13 !SAVE ^KST!</td>
</tr>
<tr>
<td>159</td>
<td>LLI</td>
<td>RR2,RR4 !CLASS!</td>
</tr>
<tr>
<td>160</td>
<td>LDL</td>
<td>RR4,KST_CLASS(R13)</td>
</tr>
<tr>
<td>161</td>
<td>CALL</td>
<td>CLASS_GF !(RR2:CLASS)!</td>
</tr>
<tr>
<td>162</td>
<td></td>
<td>!(RR4:MENTOR_CLASS)!</td>
</tr>
<tr>
<td>163</td>
<td></td>
<td>!(R1:(RET)CONDITION_CODE)!</td>
</tr>
<tr>
<td>164</td>
<td>FCN</td>
<td>R13,G15 !RESTORE PTR!</td>
</tr>
<tr>
<td>165</td>
<td>CP</td>
<td>R1,#FALSE</td>
</tr>
<tr>
<td>166</td>
<td>LDM</td>
<td>R1,@R15,#5</td>
</tr>
<tr>
<td>167</td>
<td></td>
<td></td>
</tr>
<tr>
<td>168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>!PAGE</td>
<td></td>
</tr>
</tbody>
</table>
! IF EQ THEN
LD R0, #NOT_COMPATIBLE
FISE
LDA R1, KST.MM_HANDLE(R1,3)
CALL MM_CREATE_ENTRY
!(R1:PTR TO MM_HANDLE)
!(ENTRY_NO)
!(R3:SIZE)
!(R4:CLASS)
!(R3:(RETURNED)SUCCESS_CODE)
CALL CONFINEMENT_CHECK
!(R2:SUCCESS_CODE)
FI
FI
ADD R15, #13
FI
RET
END CREATESEG
188
187
186
185
184
183
182
181
180
179
178
177
176
175
174
173
172
171
170
0286 5F0E 0292
038A 2100 0018
0431 5F0E E09F
0992 76D1 0000
0996 5F0E 0014
0998 5F0E E41A
09A2 0F0E
09A4
189
188
187
186
185
184
183
182
181
180
179
178
177
176
175
174
173
172
171
170

DELETE_SEG
PROCEDURE

191 !*****************************************************************************!
192 ! CHECKS VALIDITY OF DELETE REQUEST AND 
193 ! CALLS MM_DELETE_IF_VALID. 
194 !*****************************************************************************!
195 ! REGISTER USE: 
196 ! PARAMETERS 
197 ! R1:MENTOR_SEG NO (INPUT) 
198 ! R2:ENTRY_NO (INPUT) 
199 ! R3:SUCCESS_CODE (RETURNED) 
200 ! LOCAL USE 
201 ! R6:VARIOUS LOCAL USES 
202 !*****************************************************************************!
203
244 ENTRY
205 PUSH OR15, R1 SAVE NEEDED REGS!
206 PUSH OR15, R2
207 LD R1, #KST_SEG_NO
208 CALL ITC_GET_SEG_PTF R1: KST_SEG_NO
209 LD R13, R1 KST
210 POP R2, R15 RESTORE INPUT REGS!
211 POP R1, R13
212 SUP R1, #AR_OF_KSEGS CONVERT MENTOR_SEG NO TO
213 KST_REC_INDEX!
214 MULT RR3, #SIZEOF KST REC OFFSET TO DESIRED REC!
215 ADD R13, R1 ADD OFFSET TO KSTBASE ADDRESS!
216 LD R6, #NULL_SEG
217 CPB R16, #ST_KSEG_NO (R13)
218 IF EQ THEN IMENTOR SEGMENT NOT KNOWN
219 LD R0, #MENTOR_SEG_NOT_KNOWN
220 ELSE PUSH OR15, R1 SAVE NEEDED REGS!
221 PUSH OR15, R2
222 PUSH OR15, R13
224 !PAGE
CALL TC_GET_PROC_CLASS
I(RETURNS RA2:PROC_CLASS)
POP R13, @R15
LDL RR4,KST.CLASS(R13) IMIENTOR SEG CLASS!
PUSH @R15,R13
CALL CLASS_EQ 1( RR2:PROCESS CLASS)
1( RR4:MENTOR SEG CLASS)
1( R1:(RET) CAECTION_CODE)

LD R6,R1
PCP R13, @R15
POP R2, @R15 !RESTORE NEEDED REGS!
POP R1, @R15
CP R6,#FALSE
IF EQ THEN
LD R9,#ACCESS_CLASS_NOT_EQ
ELSIF
LDA R1,KST.MM HANDLE(R13)
CALL MM DELETE ENTRY
I(R1:MM HANDLE)
1(R2:ENTRY_NO)
1(R9:(RET)SUCCESS_CODE)
CALL CONFINEMENT_CHECK
I(R9:SUCCESS_CODE)
FI

FI
RET
END DELETE SEG
I PAGE
! MAKE_KNOWN
!********************************************************************
! CHECKS VALIDITY OF MAKE_KNOWN REQUEST AND
! CALLS MM_ACTIVATE IF VALID. ASSIGNS SEG
! NUMBER AND UPDATES KST.
!********************************************************************
!
! REGISTRER USE:
!
! PARAMETERS:
!
! R1:MENTOR_SEG_NO(INPUT)
!
! R2:ENTRY_NO(INPUT)
!
! R3:ACCESS_DESIRED(INPUT)
!
! R4:SUCCESS_CODE(RET)
!
! R1:SEGMENT_NO(RET)
!
! R2:ACCESS_ALLOWED(RET)
!
! LOCAL USE
!
! IDENTIFIED AT POINT OF USAGE
!
!********************************************************************

0110 93F1
0112 91F2
0114 2101 0002
0116 5F00 0000
0118 A10D
011A 9EF2
0120 97F1
0122 A115
0124 C4E0 E00A
0126 19D4 0210
0128 815D
012A 2104 FFFF
0130 4ADC 006E
0132 5F41 014A
0134 2103 0016
0136 2101 FFFF
253 MAKE_KNOWN
254 !********************************************************************
255 ! CHECKS VALIDITY OF MAKE_KNOWN REQUEST AND
256 ! CALLS MM_ACTIVATE IF VALID. ASSIGNS SEG
257 ! NUMBER AND UPDATES KST.
258 !********************************************************************
259 ! REGISTRER 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
271 271 PUSH @(R15,R1) !SAVE INPUT REGS!
272 272 PUSH @(R19,RR)
273 273 LD R1,E_KST_SEG_NUM
274 274 CALL ITC_GET_SEG PTR !(R1:KST_SEG_NO,RET:R0:KST)
275 275 LD R13,R3 I^KST!
276 276 POPL R2,R15
277 277 POP R1,R15
278 278 LD R5,R1 [COPY OF MENTOR SEG_NO]
279 279 SUI R5,#R_OF_KSEGS I.Convert TO INDEX!
280 280 MULT R4,#SIZEOF KST REC !KST OFFSET TO SEG REC!
281 281 ADD R13,R5 IADD OFFSET TO KST!
282 282 LD R4,#NULL_SEG
283 283 CPB R14,KST_M_SEG_NO(R13)
284 284 IF EQ THEN
285 285 LD R0,#MENTOR_SEG_NOT_KNOWN
286 286 LD R1,#NULL_SEG
287 IPAG:
LD R2,#NULL_ACCESS
ELSE
LD R7,#0 1EST INDEX!
LD R5,#NULL_SEG 1AVAIL SEG INDEX!
LD R3,R0 1'SK11!
LD R16,#NULL_SEG 1SEG KNOWN INDICATOR!
SEE IF_KNOWN:
DO
CPB R11,KST.M.SIG.NO(R9)
IF EQ THEN
CPB RL2,KST.ENTRY_NUMBER(R9)
IF EQ THEN ICASE: SEQ KNOWN!
LD R9,#SUCCESSED
ADD R7,#NR OF KSEGS
LD R1,R7 1SEG#1
LDB RL2,KST.ACCESS_MODE(R9)
LD R10,R1 1SET SEG KNOWN INDICATOR!
EXIT FROM SEE IF_KNOWN
FI
FI
CPB RL4,KST.M.SIG.NO(R9) 1SEE IF SEG # AVAILABLE!
IF E= THEN
CP R6,#NULL_SIG
IF EQ THEN
LD R3,R7 1SAVE FIRST AVAIL SEG INDEX!
ADD R5,#NR_OF_KSEGS 1CONVR10 TO SIG #!
FI
FI
INC R7!
ADD R9,#SIG2UF KST_REC 1INCREMEN1 ONE RFC!
CP R7,#MAX NO KST_PTRIPS
IF GT THEN
EXIT FROM SEE IF_KNOWN
DEN
IS_FF_KNOWN
CP R10,#NULL_SEG
IF EQ THEN ISSEG_KNOWN_INDICATOR NOT SET!
CP RE,#NULL_SEG
IF NE THEN ICASE:SEG UNKNOWN AND SEG# AVAILABLE!
PUSH @R15,RR2 ENTRY_NO &ACCESS_DESIRER!
PUSH @R15, RR2 ENTRY_NO
CALL TC.GET_ENTRY_NO ! (RET:R11:DFK_NO)!
LD R10,R1 IDBF_NO!
PCP R12,R15
POP EE,R15
PCPL RR2,R15
PCPL RR6,RH15
! MUST REEXCHANGE REGS FOR PASSING AND
RETURN CONSISTENCY OF LOCATION!
LD R5,R3 !ACCESS_DESIRER!
LD R3,R2 ENTRY_NO!
LDA R2, KST.MM HANDLE(R13) IMENTOR FPTR!
LD R6,R1 IMENTOR SEG_NO!
LD R1,PCE ISEGMENT_NO (SAVE)!
LD R4,P8 ISEGMENT_NO (PASSING ARG)!
LD R9,R6 KST!
SUB R15,#0
LLM OR15,R1,#14 ISAVE REGS 1-14!
LD R1,R10 !DFK_NO PASSED IN R1!
CALL MM ACTIVATE
! R1:DFK NO, R2:FPTR, R3:ENTRY_NO,
R4:SEGMENT_NO)!
! (RET:R0:SUCCESS CODE, RR2:CLASS, R4:SIZE)!
CALL CONFIRMATION_CHECK !(R4:SUCCESS CODE)!
LDL RH10,RR2 ICLASS!
I

01F0 A14C 01F2 1CF1 01F4 1A87 01FE C367 01 FC 1926 0200 1A7D 0202 619D 0204 5DFA 0206 6FLC 020C 0406 0210 5E2E 0214 6EFL 0216 5F30 021A 5F7D 021C 54D4 0220 5F7D 0222 9112 0224 91F4 0226 5FEE 022A 96F4 022C 9512 022E 97FD 0230 031E 0234 528E 0238 1CF1 023C 41A1 023E 7CD2 0242 5F40 0246 5F00 024A 21F1 024C 2102 0250 2107 0254 5238 0258 ECFL 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396

ID R12, R4 !SIZE! LJM R13, R15, #9 !RESTORE REGS 1-9! LD R7, R8 !SEG #1! SUB R7, #9 OF KSEGS MULT RHE, #SIZEOF KST, REC 1 OFFSET TO REC! LD R13, R7! ADD R12, R8 !ALL KST TO OFFSET! LDL KST, CLASS(R13), R10 !CLASS! LD KST, SIZE(R13), R12 !SIZE! CPB RLO, #SUCCESS! IF EQ THEN PUS E G15, R13 CALL TC GET_PROC_CLASS !(RET : RR2:PROC_CLASS)! PGP R13, G15 LDL RR4, KST, CLASS(R13) PLSH @R15, R13 PUS EL G15, RR2 PUSH @R15, R4 CALL CLASS GE !(RR2:PROC_CLASS, RR4:SEG CLASS, R13: !F1:CONDITION_CODE)! PGPL RR4, G15 POP R13, R15 POP R13, G15 CP R1, #FALSE! IF EO THEN INO ACCESS POSSIBLE--DEACT. (! LDX R1, G15, #13 LDD R1, R15 !IDT R NC! LOD R2, KST, MM_HANDLE(R15)! !HPTH! CALL MM DEACTIVATE !(RET:R0,SEG,CODE)! CALL CONFINEMENT CHECK R15,SEG,CODE! LL R1, G15 !SEG #1! LOD R2, #NULL ACCESS LDL R4,PRCC CLASS_NOT_GF,SEG CLASS ELSE PUSH G15, R13
**TERMINATE**

**PROCEDURE**

```
**---------------------------**
! CHECKS VALIDITY OF TERMINATE REQUEST!
! AND CALLS MM DEACTIVATE IF VALID
**---------------------------**

! REGISTER USE

! PARAMETERS

! R1:SEGMENT_NO (INPUT)

! R0:SUCCESS_CODE(RETURNEED)

! LOCAL USE

! R3:KST REC INDEX

! R6:CONSTANT STORAGE

! R13:~KST

**---------------------------**

ENTRY

```

02FC A113  451  
02FE 93F0  452  
02C2 1902  0010  453  
02CE 93F1  454  
02CE 93F3  455  
02CA 2101  0202  456  
02CE 5F39  0000#  457  
02D2 A12D  458  
02D4 97F3  460  
02D6 97F1  461  
02DA 913D  462  
02DF 4ADE  FFFF  463  
02E2 5E0E  02EE'  464  
02FE 2186  001C  465  
02EA 5E0E  2355'  466  
```

PAGE
02FF 2166 0001 469
02F2 4ADE 0009 470
02FE 520E 0322' 471
02FA 2164 001D 472
02FE 5E08 0338' 473
0302 #901 030A 474
0306 5F29 0312' 475
030A 210E 001E 476
030E 5E08 0338' 477
0312 93FD 478
0314 5F00 0000* 479
0318 97FD #61 480
031A 76D2 0000 482
031F 93FD 483
0320 5F00 0000* 484
0324 5F02 041A' 487
0328 97FE 489
032A 0A26 0202 490
032E 5E0E 0338' 491
0332 4CFE 04EF 492
0336 FFFF 493
033A 3E08 494
033B #497 495
033C 3EF8 496
033D 497 497
033E 498 498
033F 499 499

LD    R6,#TRUE
CPR   RL0,KST.IN_CORE(R13)
IF    EQ    THEN
LD    R6,#SEGMENT.IN_CORE
ELSE
CP    R1,#NR_OF_KSEGS
IF    LT    THEN
LD    R6,#KERNEL_SEGMENT
ELSE
PUSH  @R15,R13
CALL  TC.GET.DBR.NO
      !RETURN:RL1:DBR_NO!
POP   R13,R15
LDA   R2,KST.MM_HANDLE(R13)
PUSH  @R15,R13
CALL  MM.DEACTIVATE !(R1:DIR_NO)!
      !(R2:MM_HANDLE)!
      !(R1:SUCCESS_CODE)!
CALL  CONFINEMENT_CHECK
      !(R0:SUCCESS_CODE)!
POP   R13,R15
CPR   RL0,#SUCCeded
IF    EQ    THEN !UPDATE KST!
      LL1: KST.M.SIG_NO(R13),
      #NULL_SEG
FI
FI
FI
497
498
499
END TERMINATE
500 !PAGE
033A

SM_SWAP_IN

Procedure

501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535

ENTRY

033A A117
033C 0367 00EA
2340 1806 2210
0344 93F1
0346 93F7
2348 2101 0002
034C 5F00 0002
0350 A16D
0352 97F7
0354 97F1
0356 617D
035E 2126 FFFF
0360 4ADE 000E
0362 5F38 C36C
2364 2132 231C
0368 E838 03AA

LD R7,P1 ICOPY OF SEG #1
SUB R7,#NR OF KSEGS CONVERT SEG# TO KST_INDEX
MULT R6,#SIZECF KST_REC OFFSET TO KST_REC
PUSH @R15,R1 ISAVE SEGMENT#1
PUSH @R15,R7
LD R1,#KST_SEG_NO
CALL ITC GET SEG_PTR 1(R1:KST_SEG NO)
LD R13,R4 KST!
POP R7,R13
POP R1,R15 RETRIEVE SEGMENT#!
ADD R13,R7 IADD OFFSET TO KST EASF_ADDR!
LD RO,#NULL_SEG
CPQ RLG,KST_SEG NO(R13)
IF EQ THEN
LD R2,#SEGMENT NOT KNOWN
ELSE
PAGE
! 036C 2106 0001 536  LD R6,#TRUE
0370 4ADE 0009 537  CPR RLE,KST.IN_CORR(R13)
0374 5F0E 03E2 538  IF EQ THEN
037E 2102 2002 539  LD R2,#SUCCEEDED
0382 5F0E 03AA 540  ELSE
0386 97FD 541  PUSH R15,R13 ISAVE KST RFC ADDR!
038A 5F02 0202 542  CALL TC.GET_DBH_NO !R1:(RET)DBH_NO!
038E 97FD 543  POP R13,R15
0392 76D2 0000 544  LDA R2,KST.MM_HANDLE(R13)
0396 63DB 0026 545  LDP RL3,KST.ACCESS_MODE(R13)
039A 93FD 546  PUSH OR15,R13 ISAVE SEG KST REC ADDR!
039E 5F00 0200 547  CALL MM_SWAP IN !(R1:DBR_NO)
03A2 041A 548  !(R2:MM_HDLF)
03A6 97FD 549  !(R3:ACCESS MODE)
03AA 2002 550  !(R0:(RET)SUCCESS_CCDF)
03AC 4C15 0009 551  CALL CONFINEMENT_CHECK !(R0:SUCCESS_CODE)
03AE 0181 552  PCP R13,R15
03B2 0406 553  CPR FL0,#SUCCEEDED
03B6 5E0E 03AA 554  IF EQ THEN
03BA 4CD5 555  LDP KST.IN_CORR(R13),#TRUE
03BC 0181 556  FI
03C0 5F08 557  FI
03C4 5E08 558  RET
03C8 566 559  END MM_SWAP.IN
03CE 561 560  !PAGE2
CPB RLC,KST.IN.CORE(R13)

IF EQ THEN
LD R6,#SUCCEDED
ELSE
PUSH @R15,R13  ISAVE KST REC ADDR!
CALL TC.GET.DER.NO 1K1:(RFT)DBR.NO1
POP R13,R13
LDA R2,KST.MM.dHANDLE(R13)
PUSH @R15,R13  ISAVE SIG KST REC ADDR!
CALL MM_SWAP.OUT !(R1:DBR.NO)!

CALL CONTINEMENT_CHECK !(R0:SUCCESS_CODE)!
POP R13, @R15
CPI RLO,#SUCCEDED
IF EQ THEN
LDR &ST.IN.CORE(R13),#FALSE

FI
FI
END MM_SWAP_OUT
CONFINEMENT CHECK

621

************

622

******

623

SERVICES ROUTINE TO VERIFY CONFINEMENT IS

624

NOT VIOLATED WHEN MEM MGR SUCCESS CODE IS

625

RETURNED TO SUPERVISOR.

626

**************************************************

627

REGISTER USE:

628

PARAMETERS

629

R?: SUCCESS CODE

630

**************************************************

631

ENTRY

632

CASE #LEAF_SEG.EXISTS THEN CALL MONITOR

633

CASE #NO_LEAF_EXISTS THEN CALL MONITOR

634

CASE #ALIAS DOES NOT EXIST THEN CALL MONITOR

635

CASE #NO_CHILD_TO_DELETE THEN CALL MONITOR

636

CASE #1 AST_FULL THEN CALL MONITOR

637

CASE #2 AST_FULL THEN CALL MONITOR

638

CASE #3 AST FULL THEN CALL MONITOR

639

CASE #4 AST FULL THEN CALL MONITOR

640

PAGE
CASE #LAST_FULL THEN CALL MONITOR
CASE #LOCAL_MEMORY_FULL THEN CALL MONITOR
CASE #GLOBAL_MEMORY_FULL THEN CALL MONITOR
CASE #SEC_STOR_FULL THEN CALL MONITOR
FI
RFT
END CONFINEMENT_CHECK
END SEG_MGR
1PAGE
APPENDIX C - DISTRIBUTED MEMORY MANAGER PL/2 SYS LISTINGS

DIST_MMGR MODULE

CONSTANT
CREAT_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_DDR_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[?] BYTE]

ACTIVATE_MSG RECORD [ACTIVATE_CODE WORD
A_DDR_NO SHORT INTEGER
A_FILLER1 BYTE
A_MM_HANDLE H ARRAY
A_ENTRY_NO SHORT INTEGER
A_SEGMENT_NO SECAT_INTEGER
A_FILLER2 LONG]

DEACTIVATE_MSG RECORD [DEACTIVATE_CODE WORD
D_DDR NO SECAT 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_DRR_NO SHORT_INTEGER
SI_ACCESS_AUTH BYTE
SI_FILLER ARRAY[3 WORD]]

SWAP_OUT_MSG RECORD [SWAP_OUT_CODE WORD
SO_DRR_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
LE_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_COLL 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 [NC_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_LBR_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. invoked by segment
* manager's create segment procedure. performs type
* conversion of pointers, loads message array for
* ipc, and performs ipc with memory manager process.
* returns success_code.
*
*********************************************************************

MM_CREATE_ENTRY PROCEDURE ( HPTR ENTRY_NO ENTRY.ARRAY
   ENTRYNo SHORT INTEGER
   SIZE WORD
   CLASS LONG )

RETURNS ( SUCCESS_CODE BYTE )

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

LOCAL CB_MSPTR CB_PTR
     CM_MGBP 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!
   CB_MSPTR := CB_PTR COM_MSGPTR
   ! load arg list onto msg array!
   CB_MSGPTR := CB_MSPTR EFFECTIVE G_FRAME.
   CB_MSPTR.CB_ENTRY_NO := ENTRY_NO
   CB_MSGPTR.CB_CLASS := CLASS
   CB_MSGPTR.CB_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

! 134
**MM_DELETE_ENTRY** procedure, invoked by segment
MANAGER's delete segment procedure, performs type
conversion of pointers, loads message array for
IPC, and performs IPC with memory manager process.
Returns success code.

```
MM_DELETE_ENTRY PROCEDURE ( HPTR ENTRY_NO SECAT_INTEGER)
RETURNS ( SUCCESS_CODE BYTE )

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

LOCAL DE_MSGPT DE_PTR
     COM_MSGPTR COM_MSG
     COM_MSGPT COM.MSG
     SUC_CODE_PTR SC_PTR

ENTRY
     COM_MSGPTR := #COM_MSG
     ! TYPE CONVERSION TO OVERLAY ARG LIST ONTO MSG ARRAY !
     DE_MSGPTR := LI_PTR COM_MSGPTR
     ! LOAD ARG LIST ONTO MSG ARRAY !
     DE_MSGPTR := DELETE_ENTRY_CODE
     DE_MSGPTR.DE_MM_HANDLE[0] := HPTR[0]
     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 PROCEDEURE. INVOKED BY SEGMENT **
**MANA-**
**GER'S MAKE_ANCN PROCEDURE. PERFORMS TYPE CONVERSION**
**O2 F POINTERS, LOADS MESSAGE ARRAY FOR IPC, AND UP-**
**DATES MM_HANDLE ENTRY IN IST AFTER PERFORMING THE**
**IPC. RETURNS SUCCESS_CODE, CLASS, AND SIZE.**

**********

**MM_ACTIVATE PROCEDURE ( DER NO SECAT_INTEGER**
**H_PTR ARRAY ENTRY NO SHORT_INTEGER**
**SEGMNT NO SHORT_INTEGER )**

**RETURNS ( SUCCESS_CODE BYTE**
**CLASS LONG**
**SIZE WORD )**

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

**LOCAL A_MSGPTR A_PTR**
**COM_MSGBUF COM_MSG**
**COM_MSGPTR COM_MSG**
**RET_ARPRT 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.A_ACTIVATE_CODE := ACTIVATE_CODE**
**A_MSGPTR.A.LPR NO := DBR NO**
**A_MSGPTR.A.MM_HANDLE[0] := H_PTR[0]**
**A_MSGPTR.A.MM_HANDLE[1] := H_PTR[1]**
**A_MSGPTR.A.ENTRY NO := ENTRY NO**
**A_MSGPTR.A_SEGMENT NO := SEGMNT NO**

**PERFORM_IPC ( COM_MSGPTR )**
**! TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST !**
**RET_ARPRT := ARG_PTR COM_MSGPTR**
**SUCCESS_CODE := RET_ARPRT.R.SUC_CODE**
**CLASS := RET_ARPRT.R.CLASS**
**SIZE := RET_ARPRT.R.SIZE**
**INDEX := SEGMENT NO - NR_OF_KSEGS**
**! RETRIEVE HANDLE AND UPDATE KST !**
**KPTR := KSTPTR MM_GET_KSEG_PTR ( SEG_NO )**
**H_PTR := #KPTR.INDEX.MM_HANDLE**
**H_PTR[0] := RET_ARPRT.R.MM_HANDLE[0]**
**H_PTR[1] := RET_ARPRT.R.MM_HANDLE[1]**

**RETURN**

**END MM_ACTIVATE**

136
**MM_DEACTIVATE** procedure. Invoked by **SEG MSG**'s

**DEACTIVATE** procedure. Performs type conversion

of pointers, loads message array for IPC, and per-

forms IPC. Returns **SUCCESS_CODE**.

**DEACTIVATE** procedure. (DBR_NO SHORT_INTEGER

EPTR 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

SUCCESS_CODE_PTR SC_PTR

ENTRY

COM_MSGPTR := #COM_MSGBUF

!TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY!

D_MSGPTR := L_PTR COM_MSGPTR

!LOAD ARG LIST ONTO MSG ARRAY!

D_MSGPTR^._DEACTIVATE_CODE := DEACTIVATE_SEG_CODE

D_MSGPTR^._DBR_NO := DBR_NO


D_MSGPTR^._D_MM_HANDLE[0] := EPTR^[2]

PERFORM IPC (COM_MSGPTR)

!TYPE CONVERT TO OVERLAY MSG ARRAY ONTO RET ARG LIST!

SUCCESS_CODE_PTR := SC_PTR COM_MSGPTR

SUCCESS_CODE := SUCCESS_CODE_PTR^._SUCCESS_CODE

RETURN

END MM_DEACTIVATE
MM_SWAP_IN PROCEDURE (DBR_NO SHORT_INTEGER EPTR 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
SUCCESS_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.SI_SWAP_INFO := SWAP_IN_SEG_CODE
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!
SUCCESS_CODE_PTR := SC_PTR COM.MSGPTR
SUCCESS_CODE := SUCCESS_CODE_PTR.SUC_CODE
RETURN
END MM_SWAP_IN
**MM_SWAP_OUT** PROCEDURE (HPTR H_ARRAY, @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 MSG PTR COM MSG
COM MSGPTR CCM MSG
SUC_CODE_PTR SC_PTR

ENTRY
CCM_MSGPTR := #COM_MSGPTR
! TYPE CONVERT TO OVERLAY ARG LIST ONTO MSG ARRAY!
SO_MSGPTR := SO_PTR COM_MSGPTR
! LOAD ARG LIST INTO MSG ARRAY!
SO_MSGPTR .SWAP_OUT_CODE := SWAP_OUT_SEG_CODE
SO_MSGPTR .SO_MM_HANDLE[0] := HPTR[0]
SO_MSGPTR .SO_SVR_NO := SVR_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** procedure (DBR_NO SHORT_INTEGER) returns (DVR_VALUE ADDRESS)

* ***Note:*** Reentrant procedure **** !
* ! Save local variables on stack to ensure reentrant !

**ENTRY**

DVR_VALUE := #MM_IMAGE [DVR_NO]

**RETURN**

**END MM_GET_DBR_VALUE**

**PERFORM_IPC** procedure (CCM_MSGPTR ~CCM_MSG)

* ***Note:*** Reentrant procedure **** !
* ! Save local variables on stack to ensure reentrant !

**LOCAL** MMGR_VP_ID

**ENTRY**

1. 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_MSPTR )
   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
MAXDBG_NO := 4
KST_SEG_NO := 2
NK_OF_KSEGS := 10

TYPE

H_ARRAY ARRAY [ 3 WORD ]
COM_MSG ARRAY [ 16 BYTE ]
KST_REC RECORD [ MM_HANDLE E_ARRAY

SIZE WORD
ACCESS_MODE BYTE
IN_CORR BYTE
CLASS LONG
MSEG_NO SHORT INTEGER
ENTRY_NUMBER SHORT INTEGER ]
ADDRESS WORD

PAGE
<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>SEG_DESC_REG RECORD [BASE_ADDR ADDRESS LIMIT BYTE ATTIBUTE BYTE]</td>
</tr>
<tr>
<td>40</td>
<td>MMU RECORD [SDR ARRAY [64 SEG_DESC_REG]]</td>
</tr>
<tr>
<td>46</td>
<td>MM_VP_ID WORD</td>
</tr>
<tr>
<td>51</td>
<td>SEG_ARRAY ARRAY [MAX_SEC_SIZE BYTE]</td>
</tr>
<tr>
<td>55</td>
<td>INTERNAL</td>
</tr>
<tr>
<td>58</td>
<td>$SECTION D.MM DATA</td>
</tr>
<tr>
<td>61</td>
<td>MM_CPU_TABLE ARRAY [NO_OF_PROCESSORS MM_VP_ID] := [L]</td>
</tr>
<tr>
<td>59</td>
<td>$SECTION MSG_FRAME DCL</td>
</tr>
<tr>
<td>62</td>
<td>INOTE: THESE RECORDS ARE &quot;OVERLAYS&quot; OR &quot;FRAMES&quot; USED TO DEFINE MESSAGE FORMATS, NO MEMORY IS ALLOCATED FOR THEM !</td>
</tr>
<tr>
<td>63</td>
<td>$ABS 0</td>
</tr>
<tr>
<td>64</td>
<td>CREATE_MSG RECORD [CREATE_CODE WORD CE_MM_HANDLE E.ARRAY CE_ENTRY_NO SHORT_INTGEN CE_FILLER BYTE CE_SIZE WORD CE_CLASS LONG]</td>
</tr>
<tr>
<td>71</td>
<td>!PAGE</td>
</tr>
</tbody>
</table>
3303
72 $ABS 0
73 DELE_MSG
74 RECORD [DELETE_CODE
75 DE_MM_HANDLE L_ARRAY
76 DE_ENTRY_NO SHORT_INTEGER
77 DE_FILLER ARRAY[7 BYTE]]

0003
78 $ABS 0
79 ACTIVATE_MSG
80 RECORD [ACTIVATE_CODE
81 A_DBR_NO SHORT_INTEGER
82 A_FILLER1 BYTE
83 A_MM_HANDLE H_ARRAY
84 A_ENTRY_NO SHORT_INTEGER
85 A_SEGMENT_NO SHORT_INTEGER
86 A_FILLER2 LONG]

3303
87 $ABS 0
88 DEACTIVATE_MSG
89 RECORD [DEACTIVATE_CODE
90 D_DBR NO SHORT_INTEGER
91 D_FILLER1 BYTE
92 D_MM_HANDLE H_ARRAY
93 D_FILLER2 ARRAY[3 WORD]]

6666
94 $ABS 0
95 SWAP_IN_MSG
96 RECORD [SWAP_IN_CODE
97 SI_MM_HANDLE E_ARRAY
98 SI_DBR_NO SHORT_INTEGER
99 SI_ACCESS AUTH BYTE
100 SI_FILLER ARRAY[3 WORD]]

6666
101 $ABS 0
102 SWAP_OUT_MSG
103 RECORD [SWAP_OUT_CODE
104 SC_DFR_NC SHORT_INTEGER
105 SC_FILLER1 BYTE
106 SC_MM_HANDLE H_ARRAY
107 SC_FILLER2 ARRAY[3 WORD]]

140c 1PAGE
GLOBAL

$SECTION D_MMPROC

MM_CREATE_ENTRY PROCEDURE

******************************************************************************

! INTERFACE BETWEEN SEG MGR

! (CREATE SEG PROCEDURE) AND

! MMGR PROCESS (CREATE_ENTRY)

! PROCEDURE). ARRANGES AND

! PERFORMS IPC.

******************************************************************************

! REGISTER USE:

! PARAMETERS

! R2:SUCCESS_CODE (RET)

! R1:HPT (INPUT)

! R2:ENTRY_NO (INPUT)

! R3:SIZE (INPUT)

! RH4:CLASS (INPUT)

! LOCAL USF

! RG:MM_HANDLE ARRAY ENTRY

! RH:"COM_MSGBUF"

! R13:"COM_MSGBUF"

******************************************************************************

ENTRY

165 SUP R15,#SIZEOF COM MSG USE STACK FOR MESSAGE!

166 LD A13,R13 "COM_MSGBUF"

167 FILL COM_MSGBUF (LOAD MESSAGE). CREATE MSG FRAME

169 IS EASY AT ADDRESS ZERO. IT IS OVERLAI ONTO

170 COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADDING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!

172 CREATE_MSG.CREAE_CODE(P13),#CREATE_ENTRY_CODE

173 CREATE_MSG.CREAE_CODE(P13),#CREATE_ENTRY_CODE

174 LD RG,R1(40) ! INDEX TO MM_HANDLE ENTRY!

175 LD RG,R1(40) ! INDEX TO MM_HANDLE ENTRY!

176 !PAGE
! 0018 6FD6 0034 178 LD CREATE_MSG_CE, MM_HANDLE(1)(R13), R6
001C 3116 0004 179 LD E6, R1(#4)
0020 6FD6 0036 180 LD CREATE_MSG_CE, MM_HANDLE(2)(R13), R6
0024 6FD2 0008 181 LD CREATE_MSG_CE, ENTRY_NO(R13), R2
0028 5DD4 000C 182 LLL CREATE_MSG_CE, CLASS(R13), RR4
002C 6FD3 000A 183 LD CREATE_MSG_CE, SIZE(R13), R3
0030 A1DE 184 LD RE, R13
0032 5FD6 01A8' 185 CALL PERFORM_IPC, IA6: "COM.MSGBUF!"
0036 60DE 000C 186 LDB HL6, RFT, SUCCESS_CODE, SUCCESS_CODE(R13)
003A 310F 0010 187 ADD R15, SIZEOF COM.MSG_CE, restore stack state!
003E 9808 189 NET
0040 190 1ND MM_CREATE_ENTRY
191 !PAGE

146
```
MM_DELETE_ENTRY

******************************************************************************

! INTERFACE BETWEEN SEG MGR
! (DELETE SEG PROCEDURE) AND
! MMGR (DELETE ENTRY PROCEDURE).
! ARRANGES AND PERFORMS IPC.
!******************************************************************************

! REGISTER USE:
! PARAMETERS
! R0:SUCCESS CODE(ret)
! R1:HPTR(input)
! R2:ENTRY_NO(input)
! LOCAL USE
! R6:MM_HANDLE ARRAY ENTRY
! R8:CM_TRUNCBUF
! R13:CM_MSGBUF

******************************************************************************

ENTRY

SUB R15,#SIZEOF COM_MSGBUF USE STACK FOR MESSAGE!
LD R13,R15 !CM_MSGBUF!

!FILL CM_TRUNCBUF (LOAD MESSAGE)_DELETE_MSG FRAME
IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
ING TO EACH ENTRY) THE BASE ADDRESS OF CM_TRUNCBUF!

LD DELETE_MSG.DELETE_CODE(R13),DELETE_ENTRY_CODE
LD R6,R1(#0) INDEX TO MM_HANDLE ENTRY!
LD DELETE_MSG.DE MM HANDLE[0](R13),R6
LD R6,R1(#2)
LD DELETE_MSG.DE MMHANDLE[1](R13),R6
LD R6,R1(#4)
LD DELETE_MSG.DE MMHANDLE[2](R13),R6
LD DELETE_MSG.DE_Entry Nu(R13),R2
LE R6,R13
```
CALL PERFORM IPC IR6: ^COM_MSGBUF
IRRETIVE SUCCESS CODE FROM RETURNED MESSAGE
LDR R10,RET_SUC_CODE,SUC_CODE(R13)
ADD R15,#SIZEOF COM_MSG IRESTORE STACK STATE
END MM_DELETE_ENTRY
229C 6FD6 0038 271 LD ACTIVATE MSG.A_MMHANDLE[2](R13), R6
22A0 C6DB 003A 272 LDR ACTIVATE MSG.A_ENTRY NO(R13), R13
22A4 6EDC 003B 273 LLF ACTIVATE MSG.A_SEGMENT_NO(R13), R14
22A8 A1D8 274 LD R6, R13
22AC 93F4 275 PUSH OR15, R4 !SAVE COPY SEG #!
22AE 5F00 01A4 276 CALL PERFORM_IPC !R2:COM_MSGPUSH!
22B0 2101 00C2 277 !UPDATE KST MM_HANDLE ENTRY!
22B4 5F00 0000 278 LD R1, #KST_SEG_NO
22B8 A10C 279 CALL ITC.GET_SEG_PTR !(R1:KST_SEG_NO, RET:R0:KST)!
22BA 97F4 280 LD R12, R4 'KST'!
22BC A145 281 PCP R4, OR15 !SEG #!
22BE 03E5 00CA 282 LD R5, R4 !MOVE SEG # TO ALLOW MULT!
22F0 6014 0810 283 SUB R5, #NR_OF_KSES !CONVERT TO INDEX!
22FC B014 0210 284 MULT R4, #SIZEOF KST1_REC !OFFSET IN KST TO SEG REC!
2300 815C 285 ADD R12, R5 !ADD OFFSET TO KST!
2302 61E6 0032 286 LD R6, R.ACTIVATE ARG.R.MMHANDLE[0](R13)
2304 6FC6 0020 287 LD KST.MMHANDLE[2](R12), R6
2306 61E6 0044 288 LD R6, R.ACTIVATE ARG.R.MMHANDLE[1](R13)
2308 6FC6 0022 289 LD KST.MMHANDLE[1](R12), R6
230A 61E6 0036 290 LD R6, R.ACTIVATE ARG.R.MMHANDLE[2](R13)
230C 6FC6 0044 291 LD KST.MMHANDLE[2](R12), R6
230E 0D08 0030 292 !RETRIEVE OTHER RETURN ARGUMENTS!
2310 60D8 0038 293 LDR RLO, R.ACTIVATE ARG.R.SUC.CODE(R13)
2314 54D2 003E 294 LDL RR2, R.ACTIVATE ARG.R.CLASS(R13)
2318 61D4 003C 295 LD R4, R.ACTIVATE ARG.R.SIZE(R13)
231C 3108 0010 296 ADD R15, #SIZEOF COM MSG !RESTORE STACK STATE!
231E 9F0F 297 END MM.ACTIVATE
2320 9F0F 304 298 !PAGE
** Procedure **

** Interface between SEG MGR **

** Terminates procedure and **

** MMGR (Deactivate procedure) **

** Arranges and performs IPC **

** Register use **

** Parameters **

** R0: Success code (RET) **

** R1: Dir no (Input) **

** R2: Eptr (Input) **

** Local use **

** R6: MM HANDLE array entry **

** R6: MM MSGBUF **

** R13: MM MSGBUF **

** ENTRY **

** $15,#SIZEOF COM_MSG !USE STACK FOR MESSAGE! **

** LD R13,$15 !COM_MSGBUF! **

** FILL COM_MSGBUF (LOAD MESSAGE), DEACTIVATE MSG FRAME **

** IS BASED AT ADDRESS ZERO, IT IS OVERLAPPED WITH **

** COM_MSGBUF FRAME BY INLining EACH ENTRY (i.e. ADDING **

** TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF **

** LD DEACTIVATE_MSG,DEACTIVATE_CCDF(R13), **

** #DEACTIVATE SEG_CODE **

** LD DEACTIVATE_MSG,DEACTIVATE_CCDF(R13),R1 **

** LD R6,R2(#2) !INDEX TO MM_HANDLE ENTRY **

** LD DEACTIVATE_MSG,DEACTIVATE_CCDF[MM_HANDLE(y)(R13),R6 **

** LD R5,R2(#2) **

** LD DEACTIVATE_MSG,DEACTIVATE_CCDF[MM_HANDLE[1](R13),R6 **

** LD R6,R2(#4) **

** PAGE **
!011c cfde 0008 337  ld  deactivate_msg,d_mm_handle?(r13),r6
011c a1de 336  ld  re,r13
011c 5f00 01a8' 339  call perform_ipc !re: ~com_msgbuf!
340
341  !retrieve success_code from returned message!
342
012c 66de 0000 343  ldr  rll,ret suc_code,suc_code(r13)
2124 312f 0010 344  ald  r15,#si7xCF COM_MSG  !restore stack state!
0128 9206 345  ret
012a 346  fnd mm lfactivate
347 !page
**MM_SWAP_IN**

```
345 PROCEDURE
346
349 ! INTERFACE BETWEEN SEG_MGR (SM_!
350 ! SWAP_IN PROCEDURE) AND MMGR !
351 ! (SWAP_IN PROCEDURE) ARRANGES !
352 ! AND PERFORMS IPC. !
353 ! ********************************************
354 ! REGISTER USE:
355 ! PARAMETERS:
356 ! R0:SUCCESS CODE(RET)
357 ! R1:DPR NO(INPUT)
358 ! R2:EPTR(INPUT)
359 ! R3:ACCESS_MODE(INPUT)
360 ! LOCAL USE:
361 ! R6:MM_HANDLE ARRAY ENTRY
362 ! R5:COM_MSGBUF
363 ! R13:COM_MSGBUF
364 ! ********************************************
365
ENTRY
367 SUB R15, #SIZEOF COM_MSGBUF USE STACK FOR MESSAGE
368 LL R13, R15 ! COM_MSGBUF
369
370 IFILL COM_MSGBUF (LOAD MESSAGE). SWAP_IN_MSG FRAME
371 IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
372 COM_MSGBUF FRAME BY IN LIXING EACH ENTRY (I.E. ADD-
373 ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF
374
375 LL SWAP_IN_MSG.SWAP_IN_COLF(R13), #SWAP_IN_SEG_COLF
376
377 LL R6, R2(#0) ! INDEX TO MM_HANDLE ENTRY!
378 LL SWAP_IN_MSG.SI_MM.HANDLE[0](R13), R6
379 LL R6, R2(#2)
380 LL SWAP_IN_MSG.SI_MM.HANDLE[1](R13), R6
381 LL R6, R2(#4)
382 LL SWAP_IN_MSG.SI_MM.HANDLE[2](R13), R6
383 LL SWAP_IN_MSG.SI_DPR_lng(R13), R11
384 I PAGE
```
0152 66D9 0000 384          LDR   SWAP_IN.MSG.SI_ACCESS AUTH(R13),RL3
0156 A1DE 0000 385          LD    RE,R13
0158 5F00 01A9' 386          CALL   PERFORM_IPC IRE: ^COM_MSGBUF!
015C 6ED6 0300 387          !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
0160 010F 0010 388          LDR   RL0,RET_SUC_CODE.SUC_CODE(R13);
0164 9E2E 390          ADD    R15,#SIZEOF COM.MSG  !RESTORE STACK STATE!
0166 391          END MM_SWAP_IN
392 1PAGE
MM_SWAP_OUT

PROCEDURE

! INTERFACE BETWEEN SFG MGR (SM_!
! SWAP_OUT PROCEDURE) AND MMGR |
! (SWAP_OUT PROCEDURE) ARRANGES |
! AND PERFORMS IPC. |

! REGISTER USE: |

! PARAMETERS |

! R0: SUCCESS CODE (RET) |
! R1: ERR NO (INPUT) |
! R2: HPTR (INPUT) |

! LOCAL USE |

! R6: MM HANDLE ARRAY ENTRY |
! R7: COM.MSGBUF |
! R13: COM.MSGBUF |

**************

ENTRY

SUR R15,#SIZEOF COM.MSG IUSE STACK FOR MESSAGE!

LD R13,R15 ! COM.MSGBUF !

! FILL COM.MSGBUF (LOAD MESSAGE). SWAP_OUT_MSG FRAME
IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO |
COMM.MSGBUF FRAME BY INDEXING EACH ENTRY (I.E., ADD-
ING TO EACH ENTRY) THE BASE ADDRESS OF COM.MSGBUF !

LD SWAP_OUT.MSG.SWAP_OUT.CODE(P13), |

#SWAP_OUTSEG_CODE

LD R6,R2(#6) ! INDEX TO MM HANDLE ENTRY!

LD SWAP_OUT.MSG.SO.MM_HANDLE[0](P13),R6 |
LD R6,R2(#2) |
LD SWAP_OUT.MSG.SO.MM_HANDLE[1](R13),R6 |
LD R6,R2(#4) |
LD SWAP_OUT.MSG.SO.MM.LLHLL[2](R13),R6 |
LD SWAP_OUT.MSG.SO.DBR.NO(R13),P11

! PAGE
! 01EF A1DE 429  LD  RE,R13
0190 5F00 01AE 430  CALL  PERFORM IPC !RE: "COM MSGBUF"
0190 5F00 01AE 431  !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
0194 60DF 00CC 432  LDF  KLK,RET_SUCCESS_CODE,SUC_CODE(R13)
0192 010F 0010 433  ADD  R15,#SIZEOF COM_MSG  !RESTORE STACK STATUS!
0190 6E88 434  RET
019E 435  END MM_SWAP_OUT
01FE 436  !PAGE
! 019E 437  MM_GET_DBR_VALUE  PROCEDURE
438  ! 439  ! RETRIEVES DBR VALUE (ADDRESS
440  ! OF MMU RECPRTED TO BY DBR_)
441  ! NO)
442  ! 443  ! REGISTER USE:
444  ! 445  ! R1:DER NO (INPUT)
446  ! R2:DBR VALUE (RET)
447
448  ! 449  ENTRY
450 019E 1900 0100 449  MULT RR0,#SIZEOF MMU
451 01A2 7612 0000* 456  LDA R2,MMU_IMAGE(R1)
452 01AC 9006 451  RET
453 31A8 452  END MM_GET_DBR_VALUE
454 454 !PAGE
PROCEDURE

*****

SERVICE ROUTINE TO ARRANGE AND

PERFORM IPC WITH THE MEM MGR PROC

******

REGISTER USE:

PARAMETERS

RE: "COM_MSG(INPUT)

LOCAL USE:

R1, R2: WORK REGS

E4: "G_AST_LOCK

R13: "COM_MSGBUF

******

ENTRY

01AE 93FD
01AA 5F00 2002*
01A0 A112
C1FC 6121 0002*
C1D4 7604 0002*
C1BB 5F00 0000*
C1FC 5F00 0000*
01C0 97FD
C1C4 93FD
C1C6 5F00 0003*
C1CA 7604 0000*
C1CE 5F00 2002*
01D2 97FD
01D4 93FD
01D6

PUSH CR15,R13 !"COM_MSGBUF!
CALL IDC_GET_CPU_NO !RETR-R1:CPU_NO!
LD R2,R1
LD R1,MM_CPU_TABLE(R2), !MM_VP_ID1
LDA R4,G_AST_LOCK
CALL K LOCK
CALL SIGNAL 1R1:MM_VP_ID1,RE:"COM_MSGBUF!
PUSH CR13,R15
LD R8,R13 !"COM_MSGBUF!
CALL WAIT 1AS:"COM_MSGBUF!
LDA R4,G_AST_LOCK
CALL K_UNLOCK
POP R13,R15

END PERFORM_IPC

END DIST.MM

1PAGE
APPENDIX E - NON-DISCRETIONARY SECURITY PL/0 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
APPENDIX F - NON-DISCRETIONARY SECURITY PLZ/ASM LISTING

NDS MODULE

CONSTANT

TRUE := 1
FALSE := 3

INTERNAL

$SECTION ACC_CLASS LCL
NOTE: IS AN OVERLAY, IE NO ALLOCATION OF MEMORY!

$ABS 0

ACCESS_CLASS RECORD [LEVEL INTEGER CAT INTEGER]

GLOBAL

$SECTION NDS PRCC

CLASS_EQ PROCEDURE

ENTRY

CPL RR2,RR4
IF EQ THEN
LD R1,#TRUE
ELSE
LD R1,#1FALSE
FI

RET

END CLASS_EQ

I PAGE
CLASS GE PROCEDURE

ENTRY

PUSHL @R15,RR2 !PUSH CLASS1 ON STACK--REFER BY ADDR!

LD R13,R15 ! CLASS1 ADDR !

PUSHL @P15,RR4

LD R14,R15 ! CLASS2 ADDR !

LD R7,R14(#ACCESS CLASS.CAT) ! CAT2 IN R7 !

OR R7,ACCESS_CLASS.CAT(R13) ! CAT1 OR CAT2, R7!

CP R7,ACCESS_CLASS.CAT(R13) ! CAT1=(CAT1 OR CAT2)!

IF EQ THEN

LD R6,ACCESS_CLASS.LEVEL(R13) !LEVEL1!!

! COMPARE LEVEL1 WITH LEVEL2 !

CP R6,ACCESS_CLASS.LEVEL(R14)

IF GE THEN ! LEVEL1 GE LEVEL2 !

LD R1,#TRUE

ELSE

LD R1,#FALSE

FI

ELSE

LL R1,#FALSE

FI

POPL RR4,G15

POPL RR3,G15 !RESTORE STACK!

RET

END CLASS_GE

INL MLS

!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.

163
(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_PRB_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 Feitz's [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 AS versions be used as references; PIZ/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 signaler 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
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 MAKEKNOWN 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 MAKEKNOWN 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 e. 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.
I. 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 (9362) and the IO process (9764). 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 1 - DEMONSTRATION LISTINGS

INNER TRAFFIC CONTROL MODULF

*** 1. GETWORK:
A. NORMAL ENTRY DOES NOT SAVE REGISTERS.
   ( THIS IS A FUNCTION OF THE GATEKEEPER )
C. R14 IS AN INPUT PARAMETER TO GETWORK THAT
   SIMULATES INFO THAT WILL EVENTUALLY BE ON
   THE MMU HARDWARE. THIS REGISTER MUST BE
   ESTABLISHED AS A DBR BY ANY PROCEDURE
   INVOKING GETWORK.
D. PREEMPT INTERRUPT ENTRY HANDLER, WHICH IS
   CONTAINED IN GETWORK, DOES NOT USE THE
   GATEKEEPER AND MUST PERFORM FUNCTIONS
   NORMALLY ACCOMPLISHED BY IT
   PRIOR TO NORMAL ENTRY AND EXIT.
   ( SAVE/RESTORE: REGS, NSP; UNLOCK VPT,
   TEST INT)

2. GENERAL:
A. ALL VIOLATIONS OF
   VIRTUAL MACHINE INSTRUCTIONS ARE CONSIDERED
   ERROR CONDITIONS AND WILL RETURN SYSTEM TO
   MONITOR WITH ERROR CODE IN R0 AND PC IN R1.
B. ITM PROCEDURES CALLING GETWORK PASS DBR
   ( REGISTER R14 ) AS INPUT PARAMETER.
   ( INCLUDES: SIGNAL, WAIT, SWAP, VDPR, AND
   IDLE).

**
CONSTANT

! ************ ERROR CODES ************ !

UNAUVI LOCK := 0
MSG LIST EMPTY := 1
MSG LIST ERROR := 2
READY LIST EMPTY := 3
MSG LIST OVERFLOW := 4
SWAP NOT ALLOWED := 5
VP_INDEX_ERROR := 6

! ************ SYSTEM PARAMETERS ************ !

NR/MMC_REGS := 64 !LONG WORDS!
NR VP := 4
ILLF VP := NR VP - 1
STACK_SEG := 1
STACK_SEG_SIZE := 6100
! * * OFFSETS IN STACK_SEG * * !
STACK BASE := STACK_SEG SIZE - 140
STATUS REG BLOCK := STACK SEG SIZE - 140
F C W := STACK SEG SIZE - 140
PROCESS ID := STACK SEG SIZE - 110
N S P := STACK SEG SIZE - 110
ON := $FFFF
OFF := 0
RUNNING := 0
READY := 1
WAITING := 2
NIL := $FFFF
INVALID := $FFFFE
MONITOR := $A920 ! HBUG ENTRY !
TYPE
MESSAGE ARRAY [16 BYTE]
ADDRESS WORD
VP_INDEX INTEGER
MSG_INDEX INTEGER

MMU_TABLE_RECORD [BASE ADDRESS
ATTRIBUTES WORD]

MSG_TABLE_RECORD
[MSG MESSAGE
SENDER VP_INDEX
NEXT_MSG MSG_INDEX
FILLER ARRAY [6, WORD]
]

VP_TABLE_RECORD
{DBR ADDRESS
PRI WORD
STAF WORD
IDLE_FLAG WORD
PREEMPT WORD
PROCESSOR WORD
NEXT_READY_VP VP_INDEX
MSG_LIST MSG_INDEX
FILLER_1 ARRAY [6, WORD]
}

EXTERNAL
TC_PREEMPT_HANDLER PROCEDURE

PAGE
INTERNAL
$SECTION ITC.DAT
VPT RECORD

[ LOCK WORD
  RUNNING_LIST VP_INDEX
  READY_LIST VP_INDEX
  FREE_LIST MSG_INDEX
  FILLER_2 ARRAY [4, WORD]
  VP ARRAY [NR_VP, VP_TABLE]
  MSG_2 ARRAY [NR_VP, MSG_TABLE]
]

!PAGE
$SECTION ITC_INT_PROC

ENTRY

I TURN OFF PREEMPT_RETURN_FLAG
LD R2, #OFF

I GET STACK BASE
LD R4, R14(#STACK_SEG*4)
LDA R5, R4(#STATUS_REG_BLOCK)

I SKIP PREEMPT_HANDLER
JR END_PREEMPT_HANDLER

PREEMPT_ENTRY: ! GLOBAL LABEL !
! ** PREEMPT_HANDLER ** !
145
\[ \text{SET DRR} \]
\[ \text{LD } R2, \text{VPT.RUNNING.LIST} \]
\[ \text{LD } R14, \text{VPT.VP.DRR(R2)} \]

145
\[ \text{PUT CURRENT PROCESS IN READY STATE} \]
\[ \text{LD } \text{VPT.VP.STATE(R2), #READY} \]

150
\[ \text{SAVE ALL REGISTERS} \]
\[ \text{SUB } R15, \text{#32} \]
\[ \text{LDM } GR15, R1, \text{#16} \]

155
\[ \text{SAVE NORMAI STACK POINTER (NSP)} \]
\[ \text{LDCTL R6, NSP} \]
\[ \text{PUSH } GR15, R6 \]

157
\[ \text{GET STACK PAST} \]
\[ \text{LD } R4, R14(#STACK._SEG#4) \]
\[ \text{LDA } R5, R4(#STATUS._REG._BLOCK) \]

168
\[ \text{SAVE LAST STATUS_REGS} \]
\[ \text{LDM } R7, GR5, \text{#2} \]
\[ \text{PUSH } GR15, R7 \]
\[ \text{PUSH } GR15, R8 \]

175
\[ ! \text{PAGE} \]
! SET INTERRUPT RETURN FLAG!
LD R6, #ON
** ** ** ** ** ** **
END_PREEMPT_HANDLER:

! GET READY_VP LIST!
LD R1, VPT.READY_LIST

SELECT_VP:
DO ! UNTIL ELIGIBLE READY_VP FOUND!

CP VPT.VP.IDLE_FLAG(R1), #ON

IF EQ ! VP IS IDLE ! THEN
CP VPT.VP.PREEMPT(R1), #CN

IF EQ ! PREEMPT INTERRUPT IS ON ! THEN
EXIT FROM SELECT_VP
FI
ELSE ! VP NOT IDLE !
EXIT FROM SELECT_VP
FI

! GET NEXT READY_VP!
LD R3, VPT.VP.NEXT_READY_VP(R1)
LD R1, R3
OD

! NOTE: THE READY_LIST WILL NEVER BE EMPTY SINCE
THE IDLE VP, WHICH IS THE LOWEST PRI VP,
WILL NEVER BE REMOVED FROM THE LIST.
IT WILL RUN ONLY IF ALL OTHER READY VP'S ARE
IDLE OR IF THERE ARE NO OTHER VP'S ON
THE READY_LIST. ONCE SCHEDULED, IT
WILL RUN UNTIL RECEIVING A HDWE INTFRURPT. !
I NOTE: R14 IS USED AS DRR HERE. WHEN MMU IS AVAILABLE THIS SERIES OF SAVEF AND LOAD INSTRUCTIONS WILL BE REPLACED BY SPECIAL I/O INSTRUCTIONS TO THE MMU. !

** SAVE SP AND INTERRUPT RETURN FLAG ** !

LDM @RS, R15, #2

** SAVE FCW ** !

LDCTL R3, FCW
LD R4(#F_C_W), R3

PLACE NEW VP IN RUNNING STATE !

LD VPT.VP.STATE(P1), #RUNNING
LD VPT.RUNNING.LIST, R1

** SWAP DRR ** !

LD R14, VPT.VP.LER(R1)

LOAD NEW VP SP & INTERRUPT RET FLAG !

LD R4, R14(#STACK_SEG*4)
LDA R5, R4(#STATUS_REG_BLOCK)
LDM R15, OR5, #2

** LOAD NEW FCW ** !

LD R3, R4(#F_C_W)
LDCTL FCW, R3

TEST FOR HARDWARE INTERRUPT !

CP R0, #ON

IF EQ ! PREEMPT RETURN ! THEN

! HARDWARE PREEMPT INTERRUPT RETURN !

UNLOCK VPT !

CLR VPT.LOCK

PAGE
! TEST FOR PREEMPT !
! NOTE: SINCE A ILW INTERRUPf DOES NOT EXIT 
THROUGH THE GATE, THOSE FUNCTIONS PROVIDED 
BY A GATE EXIT TO HANDLE PREEMPTS MUST BE 
PROVIDED ELSE ALSO. !

CALL TEST_PREEMPT

1 RESTORE LAST STATUS REGS !
POP R6, @R15
POP R7, @R15
LDM @R5, R7, #2

! RESTORE NSP !
POP R6, @R15
LDCIL NSP, F6

! RESTORE ALL REGISTERS !
LDM R1, @R15, #16
ADD R15, #32

! EXECUTE HARDWARE INTERRUPT RETURN !
IREF

ELSIF ! NORMAL RETURN !
RT
FI

END GETWORK
ENTER MSG_LIST

PROCEDURE

******************************************************************************

* INSERTS POINTER TO MESSAGE FROM CURRENT VP TO SIGNALED_vp

* IN MSG_LIST

******************************************************************************

* REGISTER USE:

* PARAMETERS:

* R8(R9):MSG (INPUT)

* R1: SIGNALED VP (INPUT)

* LOCAL VARIABLES:

* R2: CURRENT_vp

* R3: FIRST_FREE_MSG

* R4: NEXT_FREE_MSG

* R5: NEXT_Q_MSG

* R6: PRESENT_Q_MSG

******************************************************************************

ENTRY

LD R2, VPT.RUNNING_LIST

!* GET FIRST MSG FROM FREE LIST !

LD R3, VPT.FREE_LIST

=* * * * DEBUG * * * * !

CP R3, #NIL

IF EQ THEN

LDA R1, $!

LD R8, #MSG_LIST OVERFLOW

CALL MONITOR

FI

!* * * END DEBUG * * * !

LD R4, VPT.MSG.Q.NEXT_MSG(R3)

LD VPT.FREE_LIST, R4

W7 !PAGE
! INSERT MESSAGE LIST INFORMATION!
LDA R12, VPT.MSG.Q.MSG(R3)
LD R5, #SIZEOF MESSAGE
LDIR @R10, @RE, R7
LD VPT.MSG.Q.SENDER(R3), R2

! INSERT MSG IN MSG LIST!
LD R5, VPT.VP.MSG_LIST(R1)
CP R5, #NIL
IF EQ ! MSG LIST IS EMPTY ! THEN
! INSERT MSG AT TOP OF LIST!
LD VPT.VP.MSG_LIST(R1), R3
ELSE ! INSERT MSG IN LIST!
MSG_Q_SEARCH:
DO WHILE NOT END OF LIST!
CP R5, #NIL
IF EQ ! END OF LIST ! THEN
EXIT FROM MSG_Q_SEARCH
FI

! GET NEXT LINK!
LD R6, R5
LD R5, VPT.MSG.Q.NEXT_MSG(R6)
OD

! INSERT MSG IN LIST!
LD VPT.MSG.Q.NEXT_MSG(R6), R3
FI
LD VPT.MSG.Q.NEXT_MSG(R3), R5
RET
END ENTER MSG_LIST
GET_FIRST_MSG

PROCEDURE

*****************************************************************************

| REMOVES MSG FROM MSG LIST |
| AND PLACES ON FREE LIST. |
| RETURNS Sender'S MSG ANL |
| VP_ID |

*****************************************************************************

REGISTER USE:

PARAMETERS:

R3(R9): MSG POINTER (INPUT)
R1: SENDFR VP (RETURNED)
LOCAL VARIABLES:
R2: CURRENT VP
R3: FIRST_MSG
R4: NEXT_MSG
R5: NEXT_FREE_MSG
R6: PROCIANT_FREE_MSG

*****************************************************************************

ENTRY

LD R2, VPT.RUNNING_LIST

! REMOVE FIRST MSG FROM MSG_LIST !
LD R3, VPT.VP.MSG_LIST(R2)

! * * * * DEBUG * * * * !
CP R3, #NIL
IF EQ TEKA
LD R2, #MSG_LIST_EMPTY
LDA R1, 0
CALL MONITOR
FI

! * * * END DEBUG * * * !
LD R4, VPT.MSG.Q.NEXT_MSG(RS)
LD VPT.VP.MSG_LIST(R2), R4
0148 F125 0336 376
014C 0505 FFFF 377
2150 5E2F 0162 378
0154 6F03 0006 379
0158 4E35 00A2 380
015C FFFF 381
015E 5E06 017E 382
0162 3305 FFFF 383
0166 5E6F 0161 384
016A 5E28 0176 385
216F A156 386
0170 6165 00A2 387
0174 FFF6 388
0178 6FC3 06A2 389
017A 6135 00A2 390
017E 6131 03A0 391
0187 763A 086E 392
018C 2127 2110 393
018A 3A11 0782 394
018F 9706 395
0190 0000 400

! INSERT MESSAGE IN FREE_LIST !
LD R5, VPT.FREE_LIST
CP R6, #NIL
IF EQ ! FREE_LIST IS EMPTY ! THEN
! INSERT AT TOP OF LIST !
LD VPT.FREE_LIST, R3
LD VPT.MSG.Q.NEXT_MSG(F3), #NIL
ELSE ! INSERT IN LIST !
FREE_Q_SEARCH:
DO
CP R5, #NIL
IF EQ ! END OF LIST ! THEN
EXIT FROM FREE_Q_SEARCH
FI
! GET NEXT MSG !
LD R6, R5
LD R5, VPT.MSG.Q.NEXT_MSG(R6)
OD

! INSERT IN LIST !
LD VPT.MSG.Q.NEXT_MSG(R6), R3
LD VPT.MSG.Q.NEXT_MSG(R3), R5
FI
! GET MESSAGE INFORMATION:
 embod SENDING_VF
LD R1, VPT.MSG.Q.SENDER(R3)
LD R10, VPT.MSG.Q.MSG(R2)
LD R7, #SIZEOF MESSAGE .
LDIR &1, OR10, OR7
RET
END GET_FIRST_MSG
MAKEROADYPROCEDURE

[***********************************]

11 INSERTSCHEDULEVPIDINTO
12 !READYLISTIAWPRIORITYAND
13 !PUTSITINREADYSTATE.
14 [***********************************]
15 !REGISTERUSE:
16 !PARAMETERS:
17 !R1:SIGNALEDVP(INPUT)
18 !LOCALVARIABLES
19 !R2:SIGVP.PRI
20 !R3:PRESENTPV
21 !R4:NEXTVP
22 [***********************************]

ENTRY

LD R3,VPTRUNNING_LIST
!* * * DEBUG * * * *
CP R4,#NIL
IF R4 !LISTISEMPTY!THEN
LD R3,#READY_LIST_EMPTY
LRA R1,0
CALLMONITOR
FI
!* *ENDDEBUG* * *

LD R2,VPVP.PRI(R1)

CP R2,VPVP.PRI(R4)
IF JTI SIGVP.PRI>READYVP.PRI!THEN
!INSERTATFRONTOFLIST!
LD VPVP.NEXTREADYVP(R1),R4
LD VPVP.READY_LIST,R1

!PAGE
L-4
E;j

ELSE ! INSERT IN LIST !

READY_LIST_SPARSE:
DO ! WHILE NOT END OF LIST !
CP R4, #NIL
IF EQ ! IF END OF LIST ! THEN
EXIT FROM READY_LIST_SEARCH
FI

CP R2, VPT.VP.PRI (R4)
IF GT ! SIG.VP.PRI > PRESENT.VP.PRI ! THEN
EXIT FROM READY_LIST_SEARCH
FI

! GET NEXT LINK !
LD R3,R4
LD R4, VPT.VP.NEXT_READY.VP(R3)
CD
I INSERT SIG.VP IN LIST !
LD VPT.VP.NEXT READY.VP(R1), R4
LD VPT.VP.NEXT READY.VP(R3), R1
FI

! CHANGE STATE TO READY !
LD VPT.VP.STATE(R1), #READY

RET
END MAKE READY
**INNER TRAFFIC CONTROL ENTRY POINTS**

GLOBAL

SECTION ITC_GLB_PROC

EALDWARE_PREEMPT LABEL

WAIT PROCEDURE

PARAMETERS

RE: MSG POINTER (INPUT)

R1: SENDING VP (RETURN)

R16: DTR (PARAM TO GETWORK)

R2: CURRENT VP (RUNNING)

R3: NEXT READY VP

R4: LOCK ADDRESS

ENTRY

LOCK VPT!

LDA R4, VPT.Lock

CALL SPIN_LOCK ! (R4: VPT.Lock)!

NOTE: RETURNS WHEN VP IS LOCKED BY THIS VP!

LD R2, VPT.RUNNING_LIST

LD R3, VPT.VP.NEXT_REALY_VP(R2)

CP VPT.VP.MSG_LIST(R2), #NIL

IF EQ ! CURRENT VP'S MSG LIST IS EMPTY ! THEN

! REMOVE CURRENT VP FROM READY LIST !

PAGE
**DEBUG**

CP R3, #NIL
IF EQ THEN
LD R3, #READY_LIST_EMPTY
LEA R1, $
CALL MONITOR
FI
! ** * END DEBUG ** * !

LD VPT, READY_LIST, R3
LD VPT, VP, NEXT_READY VP(R2), #NIL

! PUT IT IN WAITING STATE!
LD VPT, VP, STATE(R2), #WAITING

! SET DBR!
LD R14, VPT, VP, DBR(R2)
! SCHEDULE FIRST ELIGIBLE READY VP!
PUSH OR15, R6 !SAVE MSG POINTER!
CALL GETWORK 1(R14: DBR)!
PCP OR6, OR15
FI
! GET FIRST MSG ON CURRENT VP'S MSG LIST!
CALL GET, FIRST_MSG | COPIES MSG IN MSG ARRAY!
!RETURNS R1:SENDER VP!

! UNLOCK VPT!
CLR VPT, LOCK

! RETURN: R1:SENDER VP!
RET
END WAIT
SIGNAL PROCEDURE

ENTRY

LDA R4, VPT.LOCK
CALL SPIN LOCK 1 (R4:~VPT.LOCK) !

NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !

PLACE MSG IN SIGNAL VP'S MSG LIST !

CALL ENTER.MSG.LIST 1 (RE:MSG PTR, R1:SIGNALED VP) !

CP VPT.VP.STATE(R1), #WAITING

IF EJ 1 SIGNALED VP IS WAITING 1 THEN

WAKE IT UP AND MAKE IT READY !

CALL MAKE READY 1 (R1: SIGNALED VP) !

PUT CURRENT VP IN READY STATE !

LD R2, VPT.RUNNING.LIST
LD VPT.VP.STATE(R2), #READY
SET_PREEMPT PROCEDURE

SETDefaults

ENTRY

NOTE: DESIGNED AS SAFE SEQUENCE SO VPT NEED
NOT BE LOCKED.

CONVERT VP_ID TO VP_INDEX!

LEK R0, #0
MULT RR0, #SIZEOF VP_TABLE

THIS LEAVES VP INDEX IN R1!

TURN ON TGT_VP PREEMPT FLAG!

LD VPT.VP.PREEMPT(R1), #ON

** IF TARGET VP NOT LOCAL
( NOT BOUND TO THIS CPU )
[IE, IF <<CPU_SFC>>CPU_ID] VPT.VP_PHYS_CPU(P1)]
THEN SEND HARDWARE PREEMPT INTERRUPT TO
VPT.VP_CPU(R1). **!

RET

END SET_PREEMPT
**ENTRY**

LDA R4, VPT.LOCK
CALL SPIN.LOCK ! (R4:VPT.LOCK)!

! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP!

! GET CURRENT VP!
LD R2, VPT.RUNNING_LIST

! SET DBR!
LD R14, VPT.VP.DBR(R2)

! LOAD IDLE DBR ON CURRENT VP!
LL R3, #IDLE_VP=SIZECF VP_TAPLF
LD R5, VPT.VP.DBR(R3)
LD VPT.VP.DER(R2), R5

! TURN ON CURRENT VP'S IDLE FLAG!
LD VPT.VP.IDLE_FLAG(R2), #ON
I SET VP TO READY STATE!
LD VPT.VP.STATE(R2), #READY

I SCHEDULE FIRST ELIGIBLE READY VP!
CALL GETWORK I(R14: D3R)!

I UNLOCK VPT!
CLR VPT.LOCK

RIT

END IDLE
SWAP VPTR PROCEDURE

ENTRY

1 LOCK VPTR
LDA R4, VPTR.LOCK
CALL SPIN_LOCK ! (R4=VPTR.LOCK) !

1 NOTE: RETURNS WHEN VPTR IS LOCKED BY THIS VP. !
1 GET CURRENT VP !

1 R2, VPTR.RUNNING LIST
1 * * * DEBUG * * *
1 CP VPTR.VP.MSG_LIST(R2), #NIL

IF NF ! MSG WAITING ! THEN

1 R0, #SWAP NOT ALLOWED
1 LDA R1, $ ! IPC!
1 CALL MCBITCH
1 PI

1 * * END DEBUG * * !

1 SFT DIR !
LDA R14, VPTR.VP.DPP(R2)

744
701 !PA:}
IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A SECURE ARCHIVAL STORAGE

SEP W.B. J. WELLS

UNCLASSIFIED
1 LOAD NEW DBR ON CURRENT VP
LD VPT.VP.DBR(R2), R1

1 TURN OFF ILLE FLAG
LD VPT.VP.IDLE_FLAG(R2), #OFF

1 SET VP TO READY STATE
LD VPT.VP.STATE(R2), #READY

1 SCHEDULE FIRST ELIGIBLE READY VP
CALL GETWORK ! (W14:DPR) !

1 UNLOCK VPT!
CLR VPT.LOCK

RET
END SWAP_VDBR
**TEST_PREEMPT**

---

**PROCEDURE**

---

**.Bounds**

---

**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!**

---

**0106 6102 0002**

---

**010A 6121 0018**

---

**010F 0FC1 0000**

---

**0112 5E0E 011A**

---

**0116 5E06 312C**

---

**011A 6121 0018**

---

**011E 5E06 312C**

---

**0136 8E00 0000**

---

**013F 0FC1 0000**

---

**0142 5E0E 011A**

---

**0146 5E06 312C**

---

**014A 6121 0018**

---

**0160 6102 0002**

---

**016A 6121 0018**

---

**016E 5E06 312C**

---

**0176 8E00 0000**

---

**017F 0FC1 0000**

---

**0182 5E0E 011A**

---

**0186 5E06 312C**

---
! RES?T PREEMPT FLAG !
LD VPT.VP.PREEMPT(R2), #OFF

! SIMULATE PREEMPT INTERRUPT !
CALL TC_PREEMPT_HANDLER

! NOTE: THIS JUMP TO TRAFFIC CONTROL
IS USED ONLY IN THE CASE OF A PREEMPT INTERRUPT
AND SIMULATES A HARDWARE INTERRUPT. ** !

! *** END VIRTUAL PREEMPT HANDLER *** !
OD

! RETURN TO GATEKEEPER !
RET

FND TEST_PREEMPT
ENTRY

; ENTRY

! LOCK VPT!
LDA Rx, VPT.LOCK
CALL SPIN.LOCK ! (R4: VPT.LOCK)!
! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP!
LD R1, VPT.RUNNING_LIST
LEK R6, #6
! CONVET VP_INDEX TO VP_ID !
DIV RRV, #SIZEOF VP_TABLE
! * * * DPRINT * * * !
CP R0, #0
IF WE _REMAINDER <> k ! THEN
LDA R0, #VP_INDEX_ERROR
CALL MONITOR
! I
! * * * END DPRINT * * !
CLR VPT.LOCK
RET
END RUNNING_VP
SPIN_LOCK  PROCEDURE

* * * * * * * * * * *

USES SPIN LOCK MECH.
LOCKS UNLOCKED DATA
STRUCTURE (POINTED TO)
BY INPUT PARAMETER.
* * * * * * * * * * *

REGISTER USE
PARAMETERS
R4: LOCK ADDR (INPUT)
* * * * * * * * * * *

ENTRY

NOTE: SINCE ONLY ONE PROCESSOR CURRENTLY IN
SYSTEM, LOCK NOT NECESSARY. ** !
* * * * DEBUG * * * *
CP @R4, #OFF
IF NE, NOT UNLOCKED THEN
LD R6, #UNALTED_LOCK
LDA R1, $5
CALL MONITOR
FI
* END DEBUG * * !

TEST_LOCK:
BG WHILE STRUCTURE LOCKED!
TSET @R4
JR MI, TEST_LOCK
* ** NOTE: SEE PLZ/ASM MANUAL
FOR RESTRICTIONS ON
USE OF TSET. ** !

RET

END SPIN_LOCK
ITC.GET_CPU_NO

PROCEDURE

!*************************************************
! FIND CURRENT CPU_NO
! CALLED BY DIST MMGR
!*************************************************
! REGISTER USE
! R1: CPU_NO (RETURNED)
! R2: VP IMLFX (LCCAL)
!*************************************************

ENTRY

LDA R4, VPT.LOCK
CALL SPIN.Lock ! R4: ^VPT.LOCK!
LD R2, VPT.RUNNING_LIST
LD R1, VPT.VP.PHYS_PROCESSOR(R2)
CLA VPT.LOCK
RET

END ITC.GET_CPU_NO

014E
0154
0162
0176
0177
0184
0194
0200
0223
0231
0400
0520
0525
0545
0555
0577
0586
0599
8C0 ! PAGE
ENTRY

LDA R4,VPT,LOCK
CALL SPIN,LOCK !R4:~VPT,LOCK!
LD R2,VPT,RUNNING_LIST
LD R7,VPT.VP,DRR(R2)
CLR VPT,LOCK
MULT R3,#4
LD R6,R6(R1) INOT: DFR (NOT DFR,NO) USED HERE!

RET

END ITC.GET_SEG_PTR
! 01A2

K_LOCK          PROCEDURE
                !***********************************************************
                ! STUB FOR WAIT LOCK!
                !***********************************************************
                ! K4: LOCK (INPUT)!
                !***********************************************************

ENTRY
    CALL SPIN_LOCK
    RET
END K_LOCK

K_UNLOCK      PROCEDURE
                !***********************************************************
                ! STUB FOR WAIT UNLOCK!
                !***********************************************************
                ! K4: LOCK (INPUT)!
                !***********************************************************

ENTRY
    CLR  G4
    RET
END K_UNLOCK

END INNER_TRAFFIC_CONTRL

PAGE
TC MODULE

! 11 SEP 1980 !

CONSTANT
!

** ********** DEBUG CODES ********** !
FROCKED_LIST_ERROR := 4
READY_LIST_ERROR := 1
RUNNING_LIST_ERROR := 2

** ********** SYSTEM PARAMETERS ********** !
NR_PROCESSES := 4
NR_MMU_REG := 64
NP администраци := 4
NR_AVAL VP := 2
STACK SEG := 1
STACK SEG SIZE := %100

** ********** OFFSETS (FROM TOP OF STACK) ********** !
PROCESS_ID := STACK SEG SIZE-612

** ********** SYSTEM CONSTANTS ********** !
TRUE := 1
FALSE := 6
CN := %F%%
OFF := 0
RUNNING := %
READY := 1
BLOODED := 2
ILLEGAL := %DEDEL
NIL := %F%%
INVALID := %EEEE
MONITOR := %AGCZ ! DEBUG ENTRY !
TYPE
AP_POINTER WORD
ADDRESS WORD
EVENT_TABLE RECORD
[ HANDLE WORD
EVENT WORD
TICKET WORD
FILLPR_2 ARRAY [5 WORD]
]

AP_TABLE RECORD
[ DER NO SHORT_INTEGER
FILLER_3 BYTE
SAC LONG
PRI INTEGER
STATE INTEGER
NEXT_AP AP_POINTER
FILLER_1 ARRAY [2 WORD]
EVENTCOUNT EVENT_TABLE
]

RUNNING_ARRAY ARRAY [NR_AVAIL_VP WORD]
EXTERNAL
  SPIN LOCK  PROCEDURE
  SET_PREFEPT  PROCEDURE
  SWAP_WDBR  PROCEDURE
  IDLE  PROCEDURE
  RUNNING_WP  PROCEDURE
  MM_GET_DBR_VALUE  PROCEDURE
  K LOCK  PROCEDURE
  K_UNLOCK  PROCEDURE

$SECTION TC_DATA
INTERNAL
  API RECORD
  [ SUCCESS_CODE  WORD
    LOCK  WORD
    RUNNING_LIST  RUNNING_ARRAY
    READY_LIST  WORD
    BLOCKED_LIST  WORD
    FILLER  ARRAY[2 WORD]
    AP  ARRAY[NP_PROCESSES AP_TABLE]
]

GLCFAL
$SECTION TC_GLR_PROC

TC_GETWORK	PROCEDURE

******************
| LOADS NEXT READY DBR |
| ON CURRENT VP. |
******************
| REGISTER USE |
| LOCAL VARS |
| R1: CURRENT VP_ID |
| R2: READY_AP |
| R3: VP_PTR |
******************

ENTRY
! FIND FIRST READY PROCESSOR !
LD	R2, APT.REALY_LIST

READY AP SEARCH:
DC ! WHILE NOT (END OF LIST OR READY PROCESS) !

CP	R2, #NIL
IF EQ ! IF NO READY PROCESSES ! THEN
EXIT FROM READY_AP SEARCH:
FI

CP	APL.AP.STATUS(R2), #READY
IF EQ ! IF PROCESS READY ! THEN
EXIT FROM READY_AP SEARCH
FI

! GET NEXT READY AP !
LD	R3, APT.AP.NEXT_AP(R2)
LD	R2, R3
OL

PAGE
IF EQ ! IF NO PROCESSES READY ! THEN
! LOAD IDLE PROCESS !
LD APT.RUNNING.LIST(R1), #IDLE
CALL IDLE
ELSE
! LOAD FIRST READY AP !
LD APT.RUNNING.LIST(P1), R2
LD APT.AP.STATE(R2), #RUNNING
LDB R11, APT.AP.DBR_NO(R2)
CALL MP_GET_VAR_VALUE !(R1:DBR_NO)!
!(RETURN : R2:IFR)!
LD R1, R2 !DBR VALUE (ADDRESS)!
CALL SWAP_VAR !(R1:DBR)!
HI
RET
FNL TC.GFWORK
TC_PREEMPT_HANDLER PROCEDURE

ENTRY

** CALL WAIT_LOCK (APT^LOCK) **!
** RETURNS WHEN PROCESS HAS LOCKED APT **!
GET RUNNING VP ID!
CALL RUNNING_VP !(RETURNS: R1:VP ID, !

GET AP!
LD R2, APT.RUNNING_LIST(R1)

IF NOT AN IDLE PROCESS, SET IT TO READY!
CP R2, #IDLE1
IF NE ! NOT IDLE ! THEN
LD APT.AP.STATE(R2), #READY
FI

LOAD FIRST READY PROCESS!
CALL TC.GETW

** CALL WAIT_UNLOCK (APT^LOCK) **!
** RETURNS WHEN PROCESS HAS UNLOCKED APT **!
** AND ADVANCED ON THIS EVENT **!
RET
ENL TC_PREEMPT_HANDLER
TC_GET_PROC_CLASS PROCEDURE

****SEARCH IS APT FOR****
**CURRENT PROC CLASS**
**CALLED BY SFM MGR**
**PARAMETERS**
**R1:CUR VP IL(LCAL)**
**HR2:SAC(RETURNED)**
**R5:PROCESS ID(LCAL)**

ENTRY

LDA R4,APT.LOCK
CALL K_LOCK !(R4:"APT.LOCK")!
CALL RUNNING_VP !(RETURNS R1:VP.ID, !
LD R5,APT.RUNNING_LIST(R1)
LDL HR2,APT.AP.SAC(R5)
CLR APT.LOCK
LFT

END TC_GET_PROC_CLASS
TC_GET_DBG_NO

PROCEDURE

! SEARCHERS APT FOR CURRENT VALUE OF DBG_NO.

* * * * * * * * * * * * * * * * * * * * * * * * * *

! REGISTER USE:

! R1:CUR_vp_ID (LOCAL)

! R1:DBG_NO (RETURNED)

! R1:APT.LOCK (LOCAL)

! R5:PROCESS_ID (LOCAL)

* * * * * * * * * * * * * * * * * * * * * * * * * *

ENTRY

LDA R4, APT, LOCK
CALL K_LOCK ! READ APT.LOCK!
CALL RUNNING_vp 1 (RETURNS:R1:VP_ID )
LD R3, APT.RUNNING_LIST (R1)
LDR RLI, APT/AP/DBG_NO(R5)
CLR APT.LOCK
RET

END TC_GET_DBG_NO

END TC

PAGE
$SECTION IDLE_PROC

ENTRY

LD R2, #MSG

CALL WRITEIN

CALL RETURN_TO_HUG

RIT

TEST_ENTRY:

! PARAMETER R13: CALLS PROCEDURES !

CASE #3 THEN CALL IDLE

CASE #1 THEN CALL SWAP_WDE

CASE #2 THEN CALL SET_PREEMPT

CASE #3 THEN CALL TEST_PREEMPT

ELSE

CALL RETURN_TO_HUG

FI

CALL RETURN_TO_HUG

END IDLE_MAIN

FND IDLE_PROCESS

I_PAGE
2 MY_PROCESS  MODULE
3 | VERS. 1.8 |
4 |
5 CCNSANT
6 WRITE := %%EC0  ! TYYW MON CALL !
7 WRITFLN := %%FC10  ! PUTMSG MON CALL !
8 CRIP := %%FD04  ! MCRN CALL !
9 RETURN_TO_MONITOR := %A902  ! HUG RENTRY !
10 |
11 COUNT := 12
12 TIME := 500
13 |
14 SPACE := %22
15 DASH := %2D
16 |
17 IO_MGR := %28
18 FILE_MGR := %40
19 MEM_MGR := %46
20 |
21 CREATE_ENTRY_CODE := 52
22 INVALID_MGR_CODE := 60
23 DELETE_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 := 2
29 |
30 | PAGE |
31 TYPE
32 ADDRESS       WORD
33 SEG_DESC_REG RECORD [BASE_ADDR ADDRESS
34            LIMIT   BYTE
35              ATTRIBUTE BYTE]
36 MMU RECORD [SR1 ARRAY [64 SEG_DESC_REG]]
37 IR1KS_USED WORD
38 MAX_11KS WORD
39 !NOTE: LAST TWO MMU COMPONENTS LEFT
40 OFF FOR CONVENIENCE SINCE ARE NOT
41 USED FOR THE SEG MGR DEMO!
42 EXTERNAL
43 SIGNAL    PROCEDURE
44 WAIT      PROCEDURE
45
46 GLOBAL
47 $SECTION MMU_DATA
48 MMU IMAGE         ARRAY [4 MMU]
49 $SECTION MM_DATA
50 G_AST_LOCK        WORD := 0
51
52 PAGE
<table>
<thead>
<tr>
<th>53</th>
<th>INTERNAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>! * * * * MESSAGES * * * * !</td>
</tr>
<tr>
<td>55</td>
<td>IO</td>
</tr>
<tr>
<td>56</td>
<td>FM</td>
</tr>
<tr>
<td>57</td>
<td>MM_MSG_1</td>
</tr>
<tr>
<td>58</td>
<td>CREATE_MSG</td>
</tr>
<tr>
<td>59</td>
<td>DELETF_MSG</td>
</tr>
<tr>
<td>60</td>
<td>IPAGE</td>
</tr>
</tbody>
</table>
62 ACTIVATE_MSG ARRAY [* BYTE] := "&k CMM: ACTIVATE"

63 DEACTIVATE_MSG ARRAY [* BYTE] := "&k CMM: DEACTIVATE"

64 SWAP_IN_MSG ARRAY [* BYTE] := "&k CMM: SWAP_IN"

65 SWAP_OUT_MSG ARRAY [* BYTE] := "&k CMM: SWAP_OUT"

66 ERROR_MSG ARRAY [* BYTE] := "&k CMM: INVALID CODE"

E-4

$SECTION MM_PROC

ENTRY

DO ** DO FOREVER **!

LDA R2,MM_MSG_ARRAY[0]
CALL WAIT
LD SENDER, R1 !SAVE SIGNALING PROC #!
LD R3,#E
CALL MM_PRINT_BLANKS
LD R2,#MM_MSG_1
CALL WRITELN
LD R1,SENDER
IF R1
CASE #10 MGR THEN LD R2,#10

CALL WRITELN
CASE #FILE_MGR THEN LD R2,#FM

CALL WRITELN

FI
CALL MM_DELAY
CALL FILE
LD R3,#5C
CALL MM_PRINT_BLANKS
LD R1,MM_MSG_ARRAY[0]
LD E R11,MM_MSG_ARRAY[1]
<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C05C</td>
<td>0FC1</td>
<td>ELSE</td>
</tr>
<tr>
<td>0060</td>
<td>5E0E</td>
<td></td>
</tr>
<tr>
<td>0064</td>
<td>5F00</td>
<td></td>
</tr>
<tr>
<td>0068</td>
<td>5F6E</td>
<td></td>
</tr>
<tr>
<td>006C</td>
<td>0F01</td>
<td></td>
</tr>
<tr>
<td>0070</td>
<td>5E3E</td>
<td></td>
</tr>
<tr>
<td>0074</td>
<td>5F0E</td>
<td></td>
</tr>
<tr>
<td>0078</td>
<td>5F05</td>
<td></td>
</tr>
<tr>
<td>007C</td>
<td>0F01</td>
<td></td>
</tr>
<tr>
<td>0080</td>
<td>5E0E</td>
<td></td>
</tr>
<tr>
<td>0084</td>
<td>5F02</td>
<td></td>
</tr>
<tr>
<td>0088</td>
<td>5E05</td>
<td></td>
</tr>
<tr>
<td>008C</td>
<td>0F01</td>
<td></td>
</tr>
<tr>
<td>0090</td>
<td>5E0E</td>
<td></td>
</tr>
<tr>
<td>0094</td>
<td>5F00</td>
<td></td>
</tr>
<tr>
<td>0098</td>
<td>5E05</td>
<td></td>
</tr>
<tr>
<td>009C</td>
<td>0F01</td>
<td></td>
</tr>
<tr>
<td>00A0</td>
<td>5E0E</td>
<td></td>
</tr>
<tr>
<td>00A4</td>
<td>5F02</td>
<td></td>
</tr>
<tr>
<td>00AE</td>
<td>5E08</td>
<td></td>
</tr>
<tr>
<td>00AC</td>
<td>3F01</td>
<td></td>
</tr>
<tr>
<td>00E0</td>
<td>5E0F</td>
<td></td>
</tr>
<tr>
<td>00E4</td>
<td>5F20</td>
<td></td>
</tr>
<tr>
<td>00E8</td>
<td>5E08</td>
<td></td>
</tr>
<tr>
<td>00EC</td>
<td>21E2</td>
<td></td>
</tr>
<tr>
<td>00FC</td>
<td>21E2</td>
<td></td>
</tr>
</tbody>
</table>

**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_SEGMENT_CODE THEN

CALL deactivate

CASE swap_IN_SEGMENT_CODE THEN

CALL swap_IN

CASE swap_OUT_SEGMENT_CODE THEN

CALL swap_OUT

ELSE

CALL swap_CUT

LD R2,#ERROR_MSG

FI

PAGE
CALL WRITELN
CALL MM_DELAY
CALL CRIF
LD RS,#75
CALL MM_PRINT_LINE
CALL CRIF
!
** SIGNAL (SIGNALER, 'DONE') **!
LD R1, SENDER
LDA R2, MM_MSG_ARRAY[0]
CALL SIGNAL
OD ! ** REPEAT FOREVER **!
RET
END MM_MAIN

CREATE_ENTRY PROCEDURE
ENTRY
LDA R8, MM_MSG_ARRAY[0]
LDB OR8,#SUCCEEDED
LD R2,#CREATE_MSG
RET
END CREATE_ENTRY

DELETE_ENTRY PROCEDURE
ENTRY
LDA RE, MM_MSG_ARRAY[0]
LDB OR8,#SUCCEEDED
LD R2,#DELETE_MSG
RET
END DELETE_ENTRY

PAGE 3
ACTIVATE

ENTRY
LDA R8,MM_MSG_ARRAY[0]
LDA R9,RET_VALUES[3]
LD R2,#16
LDIRB @R8,@R9,R2
LD R2,#ACTIVATE_MSG
RET

END ACTIVATE

DEACTIVATE

ENTRY
LDA R5,MM_MSG_ARRAY[0]
LDA @R8,#SUCCEEDED
LD R2,#DEACTIVATE_MSG
RET

END DEACTIVATE

SWAP_IN

ENTRY
LDA R3,MM_MSG_ARRAY[0]
LD @R8,#SUCCEEDED
LD R2,#SWAP_IN_MSG
RET

END SWAP_IN
0136
185
186
187
0136 7608 0095
0136 0C35 0252
013E 2102 0071
0142 9808
0144
188
189
190
191
192
193
194
195
196
197
0144 2102 000A
0148 2101 01F4
014C 0B22 2000
0150 5E0E 0158
0154 5E5E C15F
0156 A820
015A 791D
015C 85F7
015E 9F25
216
217
218
SWAP_OUT PROCEDURE
ENTRY
LDA R2, MM_MSG_ARRAY[0]
LDR Q98,#SUCCEEDED
LD R2,#SWAP_OUT_MSG
RET
END SWAP_OUT

0144
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
MM_DELAY PROCEDURE
ENTRY
ID R2, #COUNT
LD R1, #TIME
c
CP R2, #0
IF EQ THEN EXIT FI
Dc R2
MREQ R1
OD
RFT
END MM_DELAY

PAGE
1

0164 214  MM_PRINT_LINE  PROCEDURE
0166 215  | ****************************************** |
0167 216  | PRINTS LINE LENGTH |
0168 217  | SPACE IN R3. |
0169 218  | ****************************************** |

0170 | ENTRY  |
0171 | LDF  RL6, #DASH  |
0172 | DO  |
0173 | CP  R3, #0  |
0174 | IF EQ THEN EXIT FI  |
0175 | CALL WRITE  |
0176 | DEC  R3  |
0177 | OD  |
0178 | RET  |
0179 | END MM_PRINT_LINE  |

0180 | ENTRY  |
0181 | LDA  RL6, #SPACE  |
0182 | LD  |
0183 | CP  R3, #0  |
0184 | IF EQ THEN EXIT FI  |
0185 | CALL WRITE  |
0186 | DEC  R3  |
0187 | OD  |
0188 | RET  |
0189 | END MM_PRINT_BLANKS  |
0190 | END MM_PROCESS  |
0191 | 1PAGE  |
2 FM_PROCESS MODULE
3
4 | VERS 1.8 |
5 CONSTANT
6 WRITE := \%0FC6 1 TYWR MON CALL |
7 Writeln := \%0FC0 1 PUTMSG MON CALL |
8 CPLF := \%0FD4 1 MON CALL |
9 RETURN_TO_MONITOR := \%A902 1 HRUG PREENTRY |
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 := \%60
20
21 EXTERNAL
22 SIGNAL PROCEDURE
23 WAIT PROCEDURE
24 CREATE_SEG PROCEDURE
25 DELETE_SEG PROCEDURE
26 SM_SWAP_IN PROCEDURE
27 SM_SWAP_OUT PROCEDURE
28 TERMINATE PROCEDURE
29 MAKE_KNOWN PROCEDURE
30
31 IPAGE
$SECTION FM_DATA
INTERNAL

FM_MSG_1 ARRAY [* BYTE] := "%12FM: IO = SIGNALER"

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

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

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

!PAGE
41  FM_MSG_5 ARRAY [* BYTE]:='%16FM: CALL KERNEL(SWAP_IN)'
  
42  FM_MSG_6 ARRAY [* BYTE]:='%19FM: CALL KERNEL(SWAP_OUT)'
  
43  FM_MSG_7 ARRAY [* BYTE]:='%1AFM: CALL KERNEL(TERMINATE)'
  
44  FM_MSG_8 ARRAY [* BYTE]:='%1BFM: CALL KERNEL(MAKE_KNOWN)'
  
45  !PAGE
1 00C4 46 SENDER WORD
00C6 47 FM_MSG ARRAY ARRAY [16 BYTE]
46 INTERNAL
49 $SECTION FM PROC
4C4C 50 FM_MAIN PROCEDURE
ENTRY
52 DO ** DO FOREVER **!
53 !FIRST MSG FROM IO PROC (CREATE)!
0000 5F00 0072' 54 CALL WAIT CALL
0024 5F00 0178' 55 CALL PRINT IO IS SIGNALLER
002E 5F00 0060' 56 CALL CREATE CALL
003C 5F00 0138' 57 CALL FM_PRINT_RET_FROM_KER
56 ISIGNAL IO PROC FINISHED!
2010 5F00 0064' 59 CALL FM_SIGNAL CALL
60 MSG FROM IO PROC (MK KNOWN & SWAP_IN)!
2E14 5F00 0072' 61 CALL WAIT CALL
001E 5F00 0178' 62 CALL PRINT IO IS SIGNALLER
401C 5F30 00C9' 63 CALL MK_CALL
002C 5F00 0138' 64 CALL FM_PRINT_RET_FROM_KER
0024 5F00 0104' 65 CALL SWAP_IN_CALL
0328 5F00 0138' 66 CALL FM_PRINT_RET_FROM_KER
67 ISIGNAL IO PROC FINISHED!
302C 5F00 0064' 68 CALL FM_SIGNAL CALL
69 MSG FROM IO PROC (SWAP_OUT & TERMINATE)!
0030 5F00 0072' 70 CALL WAIT CALL
0034 5F00 0178' 71 CALL PRINT IO IS SIGNALLER
603E 5F00 011B' 72 CALL SWAP_OUT_CALL
023C 5F00 0138' 73 CALL FM_PRINT_RET_FROM_KER
0340 5F00 00E4' 74 CALL TERMINATE_CALL
6144 5F00 0150' 75 CALL FM_PRINT_RET_FROM_KER
76 ISIGNAL IO PROC FINISHED!
3048 5F00 0064' 77 CALL FM_SIGNAL CALL
78 INWAIT MSG FROM IO PROC (DELETE)!
224C 5F00 0072' 79 CALL WAIT_CALL
60 I PAGE
CALL PRINT_IO, IS_SIGNALER
CALL DELETE_CALL
CALL FM_PRINT_RET_FROM_KER
ISIGNAL_IOPROC FINISHED!
CALL FM_SIGNAL_CALL
OD !***REPEAT FOREVER***!
RET
END FM_MAIN

FM_SIGNAL_CALL

ENTRY
LD R1, SENDER
LDA RE, FM_MSG_ARRAY[6]
CALL SIGNAL
RET
END FM_SIGNAL_CALL

WAIT_CALL

ENTRY
LD R8, FM_MSG_ARRAY[6]
CALL WAIT
LD SENDER, R1 ! SAVE SIGNALING PROCESS # !
RET
END WAIT_CALL
CREATE_CALL  PROCEDURE

ENTRY
LD R3,#25
CALL FM_PRINT.BLANKS
LD R2,#FM_MSG.2
CALL FM_PRINT.MSG
LD R1,#10 'MENTOR SEG #1
LD R2,#1 'ENTRY #1
LD R5,#46 'SIZE
LD R4,#3 'UPPER WORD OF CLASS
LD R5,#1 'LOWER WORD OF CLASS
CALL CREATE_SEG
RET
END CREATE_CALL

DELETE_CALL  PROCEDURE

ENTRY
LD R3,#25
CALL FM_PRINT.BLANKS
LD R2,#FM_MSG.4
CALL FM_PRINT.MSG
LD R1,#10 'MENTOR SEG #1
LD R2,#1 'ENTRY #1
CALL DELETE_SEG
RET
END DELETE CALL
MK_CALL        PROCEDURE
ENTRY          
             Entry          
LL            R3,#25
CALL          FM_PRINT_BLANKS
LD            R2,#FM_MSG.1
CALL          FM_PRINT_MSG
LD            R1,#10  !ENTRY_SEG_NO!
LD            R2,#1   !ENTRY_NO!
LD            R3,#0   !WRITE_ACCESS_DESIRED!
CALL          MAKE_KNOWN
RET            
END MK_CALL

TERMINATE_CALL PROCEDURE
ENTRY
LD            R3,#25
CALL          FM_PRINT_BLANKS
LD            R2,#FM_MSG.7
CALL          FM_PRINT_MSG
LD            R1,#11  !SEG #1!
CALL          TERMINATE
RET            
END TERMINATE_CALL
SWAP_IN_CALL PROCEDURE
ENTRY
LD R3,#25
CALL FM_PRINT_BLANKS
LD R2,#FM_MSG_5
CALL FM_PRINT_MSG
LD R1,#11 ! SEG #1
CALL SM_SWAP_IN
RET
END SWAP_IN_CALL

SWAP_OUT_CALL PROCEDURE
ENTRY
LD R3,#25
CALL FM_PRINT_BLANKS
LD R2,#FM_MSG_6
CALL FM_PRINT_MSG
LD R1,#11 ! SEG #1
CALL SM_SWAP_OUT
RET
END SWAP_OUT_CALL
FM_PRINT_RET_FROM_KER

ENTRY
CALL FM_DELAY
LD R3,#25
CALL FM_PRINT_BLANKS
LD R2,#FM_MSG 3
CALL WRITELN
CALL CR LF
LD R3,#75
CALL FM_PRINT_LINE
CALL CR LF
RET
END FM_PRINT_RET_FROM_KER

FM_PRINT_MSG

ENTRY
CALL WRITELN
CALL FM_DELAY
CALL CR LF
LD R3,#75
CALL FM_PRINT_LINE
CALL CR LF
RET
END FM_PRINT_MSG
PROCEDURE
ENTRY
LD R3, #25
CALL FM_PRINT_BLANKS
LD R2, #FM_MSG_1
CALL WRITELN
CALL CRLEF
CALL FM_DELAY
RET
FND PRINT_IO.IS_SIGNALER

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

DSC R2
MREQ R1
OD
RET
FND FM_DELAY

PAGE
01AE

01AE C92D

01B2 3F03 0300
01F4 5E0E 01BC
C1EE 5F6E C1C4
219C 5F00 0FC6
01C0 A930
C1C2 BFE6
01C4 9E85
01C6

01C6 C820

01C6 0F03 0300
01CC 5E0E 01D4
C1D0 5E0E C1DC
C1D4 5F00 0FC6
01D8 A930
C1DA BFE6
01DC 9E85
01DE

FM_PRINT_LINE

PROCEDURE

ENTRY

LD R10, #DASH
DO
CP R3, #0
IF EQ THEN EXIT FI
CALL WRITE
DEC R3
OD
RET
END FM_PRINT_LINE

FM_PRINT_BLANKS

PROCEDURE

ENTRY

LD R10, #SPACE
DO
CP R3, #0
IF EQ THEN EXIT FI
CALL WRITE
DEC R3
OD
RET
END FM_PRINT_BLANKS

END FM PROCESS
2 IO_PROCESS MODULE
3 ! VERS 1.6 !
4
5 CONSTANT
6 WRITE := %2FCE ! TYWR MON CALL !
7 WRITELN := %3FC3 ! PUTMSG MON CALL !
8 CRLF := %4FD4 ! MON CALL !
9 RETURN_TO_MONITOR := %A902 ! HHUG REENTRY !
10
11 COUNT := 16
12 TIME := 502
13
14 SPACE := %2E
15 DASH := %2E
16
17 IO_MGR := %24
18 FILE_MGR := %40
19 MEM_MGR := %36
20
21 EXTERNAL
22 MAKE_KNOWN
23 TERMINATE
24 SM_SWAP_IN
25 SM_SWAP_OUT
26 SIGNAL
27 WAIT
28 !PAGE
<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0003</td>
<td>3A</td>
<td>20 52</td>
</tr>
<tr>
<td>0006</td>
<td>45</td>
<td>41 44</td>
</tr>
<tr>
<td>0009</td>
<td>20</td>
<td>43 4F</td>
</tr>
<tr>
<td>000C</td>
<td>4D</td>
<td>4D 41</td>
</tr>
<tr>
<td>000F</td>
<td>4E</td>
<td>44</td>
</tr>
<tr>
<td>0011</td>
<td>0F</td>
<td>49 4F</td>
</tr>
<tr>
<td>0014</td>
<td>3A</td>
<td>20 53</td>
</tr>
<tr>
<td>0017</td>
<td>49</td>
<td>47 4E</td>
</tr>
<tr>
<td>041A</td>
<td>41</td>
<td>4C 20</td>
</tr>
<tr>
<td>001D</td>
<td>46</td>
<td>4D 20</td>
</tr>
<tr>
<td>0020</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>0021</td>
<td>13</td>
<td>49 4F</td>
</tr>
<tr>
<td>0024</td>
<td>3A</td>
<td>20 43</td>
</tr>
<tr>
<td>0027</td>
<td>41</td>
<td>4C 4C</td>
</tr>
<tr>
<td>002A</td>
<td>20</td>
<td>4E 45</td>
</tr>
<tr>
<td>002D</td>
<td>52</td>
<td>4E 45</td>
</tr>
<tr>
<td>0030</td>
<td>4C</td>
<td>28 4D</td>
</tr>
<tr>
<td>0033</td>
<td>41</td>
<td>4F 45</td>
</tr>
<tr>
<td>0036</td>
<td>5F</td>
<td>4B 4E</td>
</tr>
<tr>
<td>0039</td>
<td>4F</td>
<td>57 4E</td>
</tr>
<tr>
<td>003C</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>003D</td>
<td>15</td>
<td>49 4F</td>
</tr>
<tr>
<td>0040</td>
<td>3A</td>
<td>20 43</td>
</tr>
<tr>
<td>0043</td>
<td>41</td>
<td>4C 4C</td>
</tr>
<tr>
<td>0046</td>
<td>20</td>
<td>4F 45</td>
</tr>
<tr>
<td>0049</td>
<td>52</td>
<td>4E 45</td>
</tr>
<tr>
<td>044C</td>
<td>4C</td>
<td>26 53</td>
</tr>
<tr>
<td>004F</td>
<td>57</td>
<td>41 50</td>
</tr>
<tr>
<td>0052</td>
<td>5F</td>
<td>49 4E</td>
</tr>
<tr>
<td>0055</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

36 IO_MSG_3 ARRAY [* BYTE] := '1810: CALL KERNEL(SWAP_IN)'
IO_MSG_4 ARRAY [16 BYTE] := '41910: CALL KERNEL(SWAP_OUT)'

IO_MSG_5 ARRAY [16 BYTE] := '41A10: CALL KERNEL(TERMINATE)'

IO_MSG_6 ARRAY [16 BYTE] := '41610: RETURN FROM KERNEL'

IO_MSG_ARRAY ARRAY [16 BYTE]

I Paige
INTERNAL

$SECTION IO_PROC

ENTRY

DO [** DO FOREVER **!

** PRINT: "READ COMMAND" **!

LD R2, #IO_MSG 1

CALL IO_PRINT_MSG

** SIGNAL FM TO CREATE **!

AND WAIT **!

CALL IO_SIGNAL_CALL

LDA R5,IO_MSG_ARRAY[0]

CALL WAIT

LD R2, #IO_MSG 1

CALL IC_PRINT_MSG

** SIGNAL FM TO MAKE_KNOWN & SWAP_IN **!

AND WAIT **!

CALL IO_SIGNAL_CALL

LDA R5,IO_MSG_ARRAY[0]

CALL WAIT

LD R2, #IO_MSG 1

CALL IO_PRINT_MSG

IO "READ" -- MAKE_KNOWN AND SWAP_IN!

CALL IO_MK_CALL

CALL IO_SWAP_IN_CALL

LD R2, #IO_MSG 1

CALL IO_PRINT_MSG

** SIGNAL FM TO SWAP OUT AND TERMINATE!

CALL IO_SIGNAL_CALL

LDA R8,IO_MSG_ARRAY[0]

! PAGE:
0048 5F00 0300* 78
044C 21E2 0000* 79
0250 5F02 0052* 80
0254 5F06 00D6* 81
0255 5F20 00E8* 82
025C 21E2 0000* 83
0260 5F02 0082* 84
0264 5F0E 0074* 85
0268 7608 02A2* 86
006C 5F00 0000* 87
0072 9F06 90
0074 92 91
0074 95 92
0074 96 93
0074 97 94
3072 2131 0043 98
357E 760E 0011* 99
307C 5F00 0300* 100
0080 9808 101
0082 102 102
123 123
IO_PRINT_MSG  PROCEDURE

ENTRY
CALL WRITELN
CALL CR LF
CALL IO_DELAY
LD R3, #75
CALL IO_PRINT_LINE
CALL CR LF
RET
END IO_PRINT_MSG

IO_PRINT_RET_FROM_KER  PROCEDURE

ENTRY
CALL IO_DELAY
LD R2, #IO_MSG_6
CALL IO_PRINT_MSG
RET
END IO_PRINT_RET_FROM_KER

IO_MK_CALL  PROCEDURE

ENTRY
LD R2, #IO_MSG_2
CALL IO_PRINT_MSG
LD R1, #10 !MENTOR_SEG_NUMBER!
LD R2, #1 !ENTRY_NOT!
LD R3, #1 !READ_ACCESS_DESIRED!
CALL MAKE_KNOWN
RET
END IO_MK_CALL

PAGE
IO_DELAY  PROCEDURE

ENTRY
LD R2, #COUNT
LD R1, #TIME
DO
CP R2, #6
IF EQ THEN EXIT FI

FND IO_DELAY

IO_PRINT_LINE  PROCEDURE

ENTRY
LDE RLr, #DASH
DO
CP R3, #0
IF EQ THEN EXIT FI

FND IO_PRINT_LINE

242  END IO_PROCESS
243  IPAGE
LIST OF REFERENCES


<table>
<thead>
<tr>
<th>No.</th>
<th>Initial Distribution List</th>
<th>No. Copies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Defense Technical Information Center</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Cameron Station</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Alexandria, Virginia 22314</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Library, Code 0142</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93940</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Department Chairman, Code 52</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Department of Computer Science</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93940</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Lyle A. Cox, Jr., Code 52C1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Department of Computer Science</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93940</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>LTCOL Roger R. Schell, Code 52SJ</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Department of Computer Science</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93940</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Joel Trimble, Code 221</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Office of Naval Research</td>
<td></td>
</tr>
<tr>
<td></td>
<td>800 North Quincy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arlington, Virginia 2221</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>LCDR John T. Wells</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>P.O. Box 366</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Waynesboro, Mississippi 39367</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Office of Research Administration</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Code 012A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93940</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Uno R. Kodres, Code 52Kr</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Department of Computer Science</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93940</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>I. Larry Avrunin, Code 18</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DTNSRDC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bethesda, Maryland 20084</td>
<td></td>
</tr>
</tbody>
</table>

241
11. R. P. Crabb, Code 9134  
Naval Oceans Systems Center  
San Diego, California 92152

12. Kathryn Heninger, Code 7503  
Naval Research Lab  
Washington, D.C. 20375

13. Dr. J. McGraw  
U.C. - L.L.L. (L-794)  
P.O. Box 808  
Livermore, California 94550

14. Mark Underwood  
NPRDC  
San Diego, California 92152

15. Walter P. Warner, Code K70  
NSWC  
Dahlgren, Virginia 22448

16. M. George Michael  
U.C. - L.L.L. (L-76)  
P.O. Box 808  
Livermore, California 94550