THE NAVAL POSTGRADUATE SCHOOL SECURE ARCHIVAL STORAGE SYSTEM: P-ETC(U)
MAR 81 L A COX, R R SHELL, S L PERDUE

UNCLASSIFIED NPS52-81-001
The Naval Postgraduate School
SECURE ARCHIVAL STORAGE SYSTEM

Part II

- Segment and Process Management Implementation -

Lyle A. Cox, Roger R. Schell, and Sonja L. Perdue

March 1981

Approved for public release; distribution unlimited
Prepared for:
Chief of Naval Research
Arlington, Virginia 22217
This research was partially supported by grants from the Office of Naval Research, Project No. 427-001, monitored by Mr. Joel Trimble, and the Naval Postgraduate School Research Foundation.

Reproduction of all or part of this report is authorized.

This report was prepared by:

SONJA L. PERDUE
Adjunct Research Instructor of Computer Science

Reviewed by:

GORDON W. BRADLEY
Department of Computer Science

Released by:

WILLIAM M. TOLLES
Dean of Research
The security kernel technology has provided the technical foundation for highly reliable protection of computerized information. However, the operating system implementations face two significant challenges: providing (1) adequate computational resources for applications tasks, and (2) a clean, straightforward structure whose correctness can be easily reviewed. This paper presents the experience on an ongoing security kernel implementation using the Advanced Micro Devices 4116 single-board computer based on...
the Z8002 microprocessor. The performance issues of process switching, domain changing, and multiprocessor bus contention are explicitly addressed. The strictly hierarchical (i.e., loop-free) structure provides a series of increasingly capable, separately usable operating system subsets. Security enforcement is structured in two layers: the basic kernel rigorously enforces a non-discretionary (viz., lattice model) policy, while an upper layer provides the access refinements for a discretionary policy.
The Naval Postgraduate School
SECURE ARCHIVAL STORAGE SYSTEM
Part II
-Segment and Process Management Implementation-

by

Roger R. Schell, Lyle A. Cox, and Sonja L. Perdue
March 1981

Report number: NPS52-81-001
THE STRUCTURE OF A SECURITY KERNEL FOR A Z8000 MULTIPROCESSOR

LYLE A. COX, Jr., and ROGER R. SCHELL, Col., USAF

Department of Computer Science
Naval Postgraduate School
Monterey, California

ABSTRACT

The security kernel technology has provided the technical foundation for highly reliable protection of computerized information. However, the operating system implementations face two significant challenges: providing (1) adequate computational resources for applications tasks, and (2) a clean, straightforward structure whose correctness can be easily reviewed. This paper presents the experience of an ongoing security kernel implementation using the Advanced Micro Devices 4116 single-board computer based on the Z8002 microprocessor. The performance issues of process switching, domain changing, and multiprocessor bus contention are explicitly addressed. The strictly hierarchical (i.e.,
loop-free) structure provides a series of increasingly capable, separately usable operating system subsets. Security enforcement is structured in two layers: the basic kernel rigorously enforces a non-discretionary (viz., lattice model) policy, while an upper layer provides the access refinements for a discretionary policy.

**BACKGROUND**

For the last two and a half years the Naval Postgraduate School has been conducting a research and development project involving security kernel based operating systems designed for multiple processor implementations. As this work continues we feel that it is important to report on our progress and experiences, especially in the area of microprocessor implementations.

This effort has come to be known as the "SASS" or Secure Archival Storage System project [1]. In fact, this is a misnomer, as SASS is but a single instance of a more general family of secure operating systems designed early in the project [2]. While SASS has been the object of the majority of the research reported it is not the only implementation. Another operating system of this family has also been written to support a signal processing system that uses multiple Intel 8086 processors [3].

- iii -
SASS has been our principal testbed for exploring the implementation and performance issues related to these types of operating systems. SASS itself was designed to be a comprehensive multiuser, multilevel secure file storage system. As designed, it will consist of a small number of Z8000-based [4] single board computers sharing a single Multibus with storage devices and input/output devices. SASS will interface via bidirectional lines to a number of "host" systems, as illustrated in Figure 1. SASS will provide each host with a hierarchical file system. This system can be used to store and retrieve files, and share files with other hosts. This design will allow SASS to serve as a central hub of a data secure network of computers with diverse security authorization for sensitive information. SASS provides archival, shared storage while insuring that each interfaced host processor can access only that information appropriate to its security authorizations.
Figure 1: SASS System Interfaces
STRUCTURE

For this family of operating systems the security kernel technology has been used not only to effect security but also to provide the underlying organizational framework for the operating system. The SASS, one member of this family, is in the final stages of implementation. This development experience has highlighted the importance of several features that are key to this family:

- The pervasive, yet systematizing impact of the security kernel methodology [5].
- The design simplicity that accompanies a loop-free modularization that is highly compatible with the resource sharing and multiprogramming functions.
- The significance of a high degree of configuration independence, particularly for the ability to use the latest microprocessors for testbed implementation.

Independent of security, this particular kernel structure is attractive as a canonical operating system interface. It appears adequate for a wide range of functionality and capacity, and it evidences a high degree of independence from hardware idiosyncrasies. These operating system features will be discussed further below.
**Security Kernel Approach**

Members of this operating system family are organized with three distinct extended machine layers: (1) the security kernel, (2) the supervisor, and (3) the applications. This is illustrated in Figure 2. The concept of a hierarchy of extended machines is, to be sure, not new; however, the security kernel significantly constrains the organization. In particular, for reason of security all the management of physical resources must be within the kernel itself. Furthermore, confidence is increased by keeping the kernel as small and simple as possible. This means that much of what is commonly thought of as the operating system is provided outside the kernel in the supervisor layer. For this particular family member there is no major applications layer (viz., within SASS itself), since the applications are contained in the individual hosts.

The basic family of operating systems requires the kernels to provide extended virtual machines that specifically support both asynchronous processes and segmented address spaces. Within SASS, the kernel virtualizes processors, all levels of storage, and input/output. The kernel creates virtualized objects -- processes, segments, and devices. It is this "pure" virtual interface that is attractive as the
Figure 2: Extended Machine Layers
basis for canonical operating system features. The SASS supervisor is in turn built upon the kernel, using these virtualized objects to construct the file system.

Both the kernel and the supervisor have certain responsibilities for system security. The kernel manages all physical resources, and the kernel is distributed (i.e., included) in the address space of every process. At this level, isolation of the kernel -- protection from users and the supervisor -- must be provided by hardware enforced domains. The design of the system is strictly hierarchical (viz., the kernel is more privileged than the supervisor) so protection rings, as defined for Multics [6], are a satisfactory domain implementation.

The kernel has the responsibility for the enforcement of access limitations: that is, the kernel provides the mechanism for supporting non-discretionary security policy. The SASS kernel can support any such policy which can be expressed by a lattice of access classes [7]. Every object -- process, segment, or device -- has a non-forgable label that denotes its access class. This non-discretionary security has been parameterized in SASS such that exactly one module has knowledge of the interpretation of this label in terms of a specific policy. Thus, only this single module need be tailored to support a particular policy.
SASS provides discretionary security (shared access within the bounds of non-discretionary policy based on individual user identification) via the supervisor and the file structure. This discretionary security is completely outside of the kernel (in contrast with the KSOS [8] approach).

The supervisor handles the "Secure Reader-Writer Problem" with a non-exclusionary approach (one writer, retry on read) to provide synchronization between processes of different access classes. This control of interprocess communication is implemented via kernel primitives using Reed's event-counts and sequencers [9].

The SASS supervisor capabilities are achieved by associating two processes with each host link. These processes access that portion of the SASS file structure associated with that host. One of these processes provides I/O transmission and link management, while the other, a file manager, is responsible for the file system structure of its associated host. Communication between these processes (as is communication between all processes) is achieved using shared segments -- a mailbox. Synchronization is provided by the kernel (with eventcounts and sequencers).

The complementary kernel/supervisor approach to security has several advantages for SASS: the size and the complexi-
ty of the kernel can be minimized, and, given reliable host authentication, any host weaknesses will not impact the reliable enforcement of the non-discretionary security policy.

The security kernel approach constrains not only the interface but also the detailed design and implementation of internal state variables. The problem is to prevent indirect information paths between processes with different access classes. We address this problem using essentially the approach detailed by Millen [10], although without the rigor of a proof. Internal state variables, e.g., shared resource tables, are assigned an access class, and it is confirmed that its values will not be reflected to processes of an inconsistent access class. The most apparent result is that the "success code" (returned in response to the invocation of kernel primitives) primarily reflects the state of the per-process virtual resources, not the shared physical resources.
Loop Free Organization

Another aspect of the design that has helped to keep the security kernel simple and understandable is the loop-free structure of SASS. The loop-free design supports the software engineering concept of "information hiding" [11], as there are really no global data structures within SASS. The kernel is internally organized into four distinct layers, as illustrated in Figure 3; these layers, that will be described below, are termed (1) segment and event managers, (2) traffic controller, (3) memory manager, and (4) inner traffic controller.

In practice we have been quite doctrinaire in enforcement of the loop-free structure for this organization. While many operating systems claim to be modular or well-structured, we empirically validate this claim. We "peel-off" the upper layers one at a time by literally removing the code and data, and then demonstrate that the remainder can be loaded and run as a functionally intact, but obviously limited, operating system subset. The function of each layer will now be described, proceeding from the bottom upward.
Figure 3: Internal Kernel Organization
Inner Traffic Controller. Processor multiplexing has two layers, similar to those proposed for Multics [12]. Each physical processor has a fixed number of "virtual processors" that are multiplexed onto it. Two of these virtual processors are dedicated to system services: an idle virtual processor and a memory manager process to manage the asynchronous access to secondary storage devices. The remaining virtual processors (currently two per physical processor) are available to the (upper level) traffic controller. The inner traffic controller provides signal and wait synchronization primitives that include a message that is passed between virtual processors. In terms of traditional jargon, the inner traffic controller provides multiprogramming by scheduling virtual processors to run on the CPU they are (permanently) associated with. Note that this structure implies that the security kernel is interruptible, viz., is not a critical section; however, the inner traffic controller itself is not interruptible. In addition, this layer provides all the multiprocessing interactions between individual physical processors, using a hardware "preempt" interrupt.

Memory Manager. This layer manages the multiplexing of the physical storage resources, viz., "disk" and "core". This layer also manages the segment descriptors in the memory management unit (MMU) image for each process. Most of the functions of this layer are executed by the per-CPU memory
manager processes, with synchronization provided by inner traffic controller signal and wait primitives. The single board computers have per-processor, local memory; there is also additional global memory that is addressable by all processes. The memory manager insures that (only) shared segments are in global memory.

This policy can require some transfers between local and global memory; however, the low transaction rate of the archival storage system is not demanding, and this structure minimizes bus transfer requirements under expected operating conditions.

Traffic Controller. The variable number of processes (two per host) are multiplexed onto virtual processors defined by the inner traffic controller. Each process has an affinity to the physical processor whose local memory contains a portion of its address space at the time of the process scheduling decision. As indicated earlier, the traffic controller layer uses Reed's advance and await mechanism [9] to provide interprocess communication.

Segment and Event Managers. All entries into the kernel pass through the segment/event manager layer. The explicit non-discretionary security checks are made at this level by comparing the access class labels of subjects and objects. This layer uses a per-process known segment table to convert
process local names (viz., segment number) for objects into system-wide names. Each segment has associated with it two eventcounts and a sequencer; thus, segment numbers also serve as their names. The segment manager provides for the creation and deletion of segments and their entry into and removal from a process address space.

*Gate Keeper.* A process invokes a security kernel function using the traditional trap mechanism. The 28000 "system call" instruction causes a trap, and the gate keeper is merely the trap handler. All parameters and return values are "passed by value" in CPU registers; this simplifies security validation. The gate keeper merely calls the particular procedure that corresponds to the requested function.

**Microprocessor Testbed**

One important aspect of this research has been the actual implementation and testing of the concepts developed. Traditionally the implementation of multiple processor structures has been an expensive undertaking. Recently the development of sophisticated microprocessors that feature multiple operating modes, advanced addressing, support of multiple processor configurations, and a standard bus configuration with peripheral support have all made the implementation of advanced operating systems on microprocessor devices possible, and economically feasible.
The processors of SASS all share the same bus; each processor is a commercial single board computer with on-board random access memory. These processors also share a global memory, and certain peripheral devices. This configuration is illustrated in Figure 4.

In general, security kernel based operating systems find three processor-supported execution domains (operating states) highly desirable: for the kernel, supervisor, and applications. This is true of the operating system family discussed here. Currently there are no single chip processors that support three states. This is not a significant problem for SASS, since it is the hosts rather than the SASS system processors that execute user application programs. Under these circumstances a two mode (kernel and supervisor) machine is sufficient. Such architectures are currently available as microprocessors, in particular the Z8000.

Accordingly, we are implementing a multiple microprocessor system to test the SASS concept. The current hardware in use is the AMD 4116 single board computer [13] in a standard Multibus backplane. This configuration has a significant limitation: it does not include the hardware Memory Manager Unit, as described in [2].
Figure 4: Multiprocessor Configuration
Currently we simulate in software the memory management unit, so the kernel is not protected from the supervisor as the original design specified. Hardware protection in the form of addressing limitations is available, and has been used in some of the experiments to assure the integrity of the kernel. In this configuration, the hardware protects one half of the local memory from any access when the CPU is operating in the normal mode. Any attempt to access memory which is thus protected generates an interrupt and the fault detection software traps the access. This is adequate for current tests, but a complete memory management system is clearly more desirable. Our experiences on this testbed in terms of performance and software development are discussed further below.

THE SASS EXPERIENCE

The lessons learned to this point fall into two broad categories: programming (software engineering) experiences, and performance experiences. We will discuss both of these issues below.

Programming Experiences

The nature of this research effort has been highly structured, emphasizing modularity at every opportunity. The software design is strictly "top-down." This has been a
matter of good design practice, and of necessity. Since the majority of the work has been performed by a succession of Master's degree students \([14,15,16,17,18,19]\) during their brief six to nine months of research each, the clear definition of software modules has been key to the success of the effort. We have found that the high degree of modularity has allowed the students to work on the project with a minimum of "start-up" time, and a maximum of productive effort and learning.

The actual implementation is proceeding in an essentially bottom up manner, with test harnesses and stubs being written as necessary for testing. The SASS modules were specified in a pseudo-language resembling current higher level languages. The SASS modules as implemented were coded in PLZ-AS3 \([20]\), the Z8000 structured assembly language. We found that the pseudo-code specifications of modules were adequate, and that the translation from this code to the structured assembly language was straightforward.

The structured assembly language of the Zilog Z8000 supported many of the constructs usually thought to be unique to higher level languages, including typed record structures, DO-loops, IF-THEN-ELSE, and CASE. In fact, our programmers think of this assembly language as a higher level language. Approximately 40 percent of the statements writ-
ten in SASS are equivalent to statements in modern programming languages.

Despite the qualities of the structured assembler, it was selected by default. When the decision was made, the prototype hardware boards were just becoming available. There was virtually no software support available. In particular, no higher level language was available. The software environment was (by modern standards) very primitive, with no tools for operating system development available. Nevertheless, the progression from microprocessor development system to commercial single board computer system has been surprisingly smooth (an opinion that some students might dispute). The software development environment has grown slowly. Yet, this has not proved to be a handicap.
Performance Issues

In the programming for the SASS, we have generally treated performance as a secondary issue, in deference to more basic concerns such as security and modularity. However, we have addressed performance on a design level where performance is strongly related to architectural choices.

Obviously, one basic design choice is the use of multiprocessing as a way to increase processing capacity. However, bus contention is a major performance concern in the multiprocessor configurations, since all processors share a single Multibus. If, for example, all code and data were located in global memory, then even two or three processors would saturate the bus. However, in reality only shared, writable segments need be in global memory. Our use of a purely virtual, segmented memory permits the kernel to determine exactly which are the shared, writable segments. As noted before, the memory manager layer totally controls the allocation to global memory, and thus markedly controls bus contention.

In the current SASS implementation we use the "Normal" and "System" modes of the Z8000 hardware, with the system mode dedicated to the security kernel. The domain changes automatically generate a switch of the stack within the
hardware. This is particularly important to the efficiency with which we can switch domains while maintaining the integrity of the kernel.

In SASS a process switch is achieved by switching the stack. SASS saves the process history in the stack, so a switch requires only the stack exchange. Preempt hardware interrupts can initiate scheduler changes, and associated virtual interrupts to the virtual processors. This sequence is relatively efficient given the Zilog architecture. The process switching performance question is more interesting in the context of processor multiplexing.

The multiprogramming time is the interval from the time the inner traffic controller signal primitive is invoked in one virtual processor until there is a return from a (pending) wait invocation in a different virtual processor. This includes both process switching and message passing operations.

For interprocess communication, the read and ticket calls (from the normal mode) include a system call though the gatekeeper to the kernel, the non-discretionary security checks, and access to the eventcount or sequencer value; however, no process switch is involved. The synchronization time includes the interval from the invocation of the system call
(in normal mode) for advance in one process until the return from a (blocking) await invocation in a different process. This includes the security checks and scheduling of both a virtual and a physical processor.

A set of measurements on the current implementation are summarized in Table 1. There has been no effort to "tune" the system to improve performance. We find these results within our range of expectations for a single chip microprocessor.

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprogramming signal/wait pair</td>
<td>0.5</td>
</tr>
<tr>
<td>Synchronization advance/await pair</td>
<td>2.3</td>
</tr>
<tr>
<td>Read (Eventcount)</td>
<td>0.6</td>
</tr>
<tr>
<td>Ticket (Sequencer)</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 1. Performance Measurements
A modern operating system featuring kernel-based security, segmented memory, and multiple processors has been designed and is being implemented using modern microprocessors. To date our focus on methodical design has paid off: the implementation of a carefully designed, simple structure using elementary software development tools has proceeded well.

The initial testbed implementation is running and preliminary data is now available regarding the operating performance of such systems implemented on microprocessors of advanced architectures. Data gathered suggests that the security kernel is indeed an attractive structure for a modern operating system. There is a wide range of applications where sophisticated operating systems can be implemented upon microprocessors, and attractive performance can be achieved, particularly through the use of multiple processors.
ACKNOWLEDGMENTS

The authors would like to acknowledge the many long hours of work and dedicated effort contributed by E. E. Moore, A. V. Gary, S. L. Reitz, J. T. Wells, and A. R. Strickler, the students of the SASS project. Without their dedication, ideas and effort, this project would never have been able to progress. Specifically, we would like to acknowledge the contributions of Ms. C. Yamanaka and Ms. N. Seydel, whose typing and assistance were invaluable. This research was partially supported by grants from the Office of Naval Research, Project No. 427-001, monitored by Mr. Joel Trimble, and the Naval Postgraduate School Research Foundation.
REFERENCES


FOREWORD

This technical report contains edited segments of four masters' theses:

The Design and Implementation of the Memory Manager for a Secure Archival Storage System by E. E. Moore and A. V. Gary


Implementation of Segment Management for a Secure Archival Storage System by J. T. Wells

Implementation of Process Management for a Secure Archival Storage System by A. R. Strickler

which describe the development and implementation of the Naval Postgraduate School Secure Archival Storage System (SASS). These theses are based upon the design outlined in the Naval Postgraduate School SECURE ARCHIVAL STORAGE SYSTEM Part I - Design - by R. R. Schell and L. A. Cox [17]. This design is updated and presented in detail.

Some sections of each thesis have been excluded in order to eliminate repetition and bulk. Similarly, the program listings in this report represent the current state of the project and do not pertain to any one thesis. An attempt has been made to footnote some discrepancies between the
system described by these theses and the current state. However, there may be some details described herein which do not correspond to the current SASS system. Consequently, the reader is advised to consult the individual thesis if more detail on a particular phase of the development is required. A program description document, providing greater clarification of SASS organization and listings, is also available.
# CONTENTS

THE STRUCTURE OF A SECURITY KERNEL FOR A Z8000 MULTIPROCESSOR ................................... ii

FOREWORD .......................................................... XXX

PART A -- INTRODUCTION

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. BACKGROUND</td>
<td>2</td>
</tr>
<tr>
<td>II. BASIC CONCEPTS/DEFINITIONS</td>
<td>6</td>
</tr>
<tr>
<td>PROCESS</td>
<td>6</td>
</tr>
<tr>
<td>INFORMATION SECURITY</td>
<td>8</td>
</tr>
<tr>
<td>SEGMENTATION</td>
<td>13</td>
</tr>
<tr>
<td>PROTECTION DOMAINS</td>
<td>15</td>
</tr>
<tr>
<td>ABSTRACTION</td>
<td>16</td>
</tr>
</tbody>
</table>

PART B -- SECURE ARCHIVAL STORAGE SYSTEM DESIGN

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>III. BASIC SASS OVERVIEW</td>
<td>18</td>
</tr>
<tr>
<td>IV. SUPERVISOR</td>
<td>23</td>
</tr>
<tr>
<td>FILE MANAGER PROCESS</td>
<td>24</td>
</tr>
<tr>
<td>INPUT/OUTPUT PROCESS</td>
<td>25</td>
</tr>
<tr>
<td>V. GATE KEEPER</td>
<td>26</td>
</tr>
<tr>
<td>VI. DISTRIBUTED KERNEL</td>
<td>28</td>
</tr>
<tr>
<td>SEGMENT MANAGER</td>
<td>29</td>
</tr>
<tr>
<td>EVENT MANAGER</td>
<td>32</td>
</tr>
<tr>
<td>NON-DISCRETIONARY SECURITY MODULE</td>
<td>33</td>
</tr>
<tr>
<td>TRAFFIC CONTROLLER</td>
<td>33</td>
</tr>
<tr>
<td>INNER TRAFFIC CONTROLLER</td>
<td>38</td>
</tr>
<tr>
<td>DISTRIBUTED MEMORY MANAGER</td>
<td>41</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>XIII. INTRODUCTION</td>
<td>109</td>
</tr>
<tr>
<td>XIV. IMPLEMENTATION</td>
<td>112</td>
</tr>
<tr>
<td>DEVELOPMENTAL SUPPORT</td>
<td>112</td>
</tr>
<tr>
<td>INNER TRAFFIC CONTROLLER</td>
<td>114</td>
</tr>
<tr>
<td>Virtual Processor Table (VPT)</td>
<td>114</td>
</tr>
<tr>
<td>Level-1 Scheduling</td>
<td>118</td>
</tr>
<tr>
<td>Getwork</td>
<td>119</td>
</tr>
<tr>
<td>Virtual Processor Instruction Set</td>
<td>126</td>
</tr>
<tr>
<td>Wait</td>
<td>127</td>
</tr>
<tr>
<td>Signal</td>
<td>130</td>
</tr>
<tr>
<td>SWAP_VDBR</td>
<td>131</td>
</tr>
<tr>
<td>IDLE</td>
<td>133</td>
</tr>
<tr>
<td>SET_VPREEMPT</td>
<td>134</td>
</tr>
<tr>
<td>TEST_VPREEMPT</td>
<td>135</td>
</tr>
<tr>
<td>TRAFFIC CONTROLLER</td>
<td>138</td>
</tr>
<tr>
<td>Active Process Table (APT)</td>
<td>138</td>
</tr>
<tr>
<td>Level-2 Scheduling</td>
<td>141</td>
</tr>
<tr>
<td>TC_GETWORK</td>
<td>141</td>
</tr>
<tr>
<td>TC_PREEMPT_HANDLER</td>
<td>143</td>
</tr>
<tr>
<td>Eventcounts</td>
<td>145</td>
</tr>
<tr>
<td>Advance</td>
<td>145</td>
</tr>
<tr>
<td>Await</td>
<td>146</td>
</tr>
<tr>
<td>Read</td>
<td>146</td>
</tr>
<tr>
<td>Ticket</td>
<td>146</td>
</tr>
<tr>
<td>SYSTEM INITIALIZATION</td>
<td>147</td>
</tr>
<tr>
<td>XIV. CONCLUSION</td>
<td>151</td>
</tr>
<tr>
<td>RECOMMENDATIONS</td>
<td>151</td>
</tr>
<tr>
<td>FOLLOW ON WORK</td>
<td>152</td>
</tr>
<tr>
<td>PART E -- IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A</td>
<td></td>
</tr>
<tr>
<td>SECURE ARCHIVAL STORAGE SYSTEM</td>
<td></td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>XVI. INTRODUCTION</td>
<td>154</td>
</tr>
<tr>
<td>XVII. SEGMENT MANAGEMENT FUNCTIONS</td>
<td>155</td>
</tr>
<tr>
<td>SEGMENT MANAGER</td>
<td>155</td>
</tr>
<tr>
<td>Function</td>
<td>155</td>
</tr>
<tr>
<td>Database</td>
<td>156</td>
</tr>
<tr>
<td>NON-DISCRETIONARY SECURITY MODULE</td>
<td>160</td>
</tr>
<tr>
<td>MEMORY MANAGER</td>
<td>161</td>
</tr>
<tr>
<td>Function</td>
<td>161</td>
</tr>
<tr>
<td>Databases</td>
<td>162</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>XX. INTRODUCTION</td>
<td>196</td>
</tr>
<tr>
<td>XXI. IMPLEMENTATION ISSUES</td>
<td>198</td>
</tr>
<tr>
<td>DATABASE INITIALIZATION</td>
<td>198</td>
</tr>
<tr>
<td>Inner Traffic Controller Initialization</td>
<td>199</td>
</tr>
<tr>
<td>Traffic Controller Initialization</td>
<td>202</td>
</tr>
<tr>
<td>Additional Initialization Requirements</td>
<td>205</td>
</tr>
<tr>
<td>PREEMPT INTERRUPTS</td>
<td>206</td>
</tr>
<tr>
<td>Physical Preempt Handler</td>
<td>207</td>
</tr>
<tr>
<td>Virtual Preempt Handler</td>
<td>209</td>
</tr>
<tr>
<td>IDLE PROCESSES</td>
<td>213</td>
</tr>
<tr>
<td>ADDITIONAL KERNEL REFINEMENTS</td>
<td>215</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>217</td>
</tr>
<tr>
<td>XXII. PROCESS MANAGEMENT IMPLEMENTATION</td>
<td>218</td>
</tr>
<tr>
<td>EVENT MANAGER MODULE</td>
<td>220</td>
</tr>
</tbody>
</table>
Support Procedures ........................................ 221
Read .......................................................... 223
Ticket ......................................................... 223
Await .......................................................... 224
Advance ........................................................ 225

TRAFFIC CONTROLLER MODULE ............................... 225
TC_Network .................................................... 226
TC_Await ....................................................... 227
TC_Advance ..................................................... 228
Virtual_Preempt_Handler .................................. 233
Remaining Procedures ....................................... 233

DISTRIBUTED MEMORY MANAGER MODULE ................. 234
MM_Read_Eventcount ....................................... 235
MM_Advance .................................................... 235
MM_Ticket ....................................................... 236
MM_Allocate .................................................... 236

GATE_KEEPER MODULES ...................................... 238
User_Gate Module ........................................... 239
Kernel_Gate_Keeper Module ............................... 242

SUMMARY ...................................................... 243

IXIII. CONCLUSION ........................................... 244

FOLLOW ON WORK ............................................ 245

Appendix

<table>
<thead>
<tr>
<th>A. EVENT MANAGER LISTINGS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B. TRAFFIC CONTROLLER LISTINGS</td>
<td>247</td>
</tr>
<tr>
<td>C. DISTRIBUTED MEMORY MANAGER LISTINGS</td>
<td>258</td>
</tr>
<tr>
<td>D. GATE_KEEPER LISTINGS</td>
<td>287</td>
</tr>
<tr>
<td>E. BOOTSTRAP_LOADER LISTINGS</td>
<td>308</td>
</tr>
<tr>
<td>F. LIBRARY FUNCTION LISTINGS</td>
<td>317</td>
</tr>
<tr>
<td>G. INNER TRAFFIC CONTROLLER LISTINGS</td>
<td>330</td>
</tr>
<tr>
<td>H. SEGMENT MANAGER LISTINGS</td>
<td>335</td>
</tr>
<tr>
<td>I. NON-DISCRETIONARY SECURITY LISTINGS</td>
<td>364</td>
</tr>
<tr>
<td>J. MEMORY MANAGER LISTINGS</td>
<td>385</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>387</td>
</tr>
<tr>
<td></td>
<td>404</td>
</tr>
</tbody>
</table>

- xxxvi -
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>SASS System Interfaces</td>
<td>v</td>
</tr>
<tr>
<td>2.</td>
<td>Extended Machine Layers</td>
<td>viii</td>
</tr>
<tr>
<td>3.</td>
<td>Internal Kernel Organization</td>
<td>xiii</td>
</tr>
<tr>
<td>4.</td>
<td>Multiprocessor Configuration</td>
<td>xviii</td>
</tr>
<tr>
<td>5.</td>
<td>SASS System</td>
<td>20</td>
</tr>
<tr>
<td>6.</td>
<td>System Overview (Dual Host)</td>
<td>22</td>
</tr>
<tr>
<td>7.</td>
<td>Known Segment Table (KST)</td>
<td>31</td>
</tr>
<tr>
<td>8.</td>
<td>Active Process Table (APT)</td>
<td>35</td>
</tr>
<tr>
<td>9.</td>
<td>Virtual Processor Table (VPT)</td>
<td>39</td>
</tr>
<tr>
<td>10.</td>
<td>Extended Instruction Set</td>
<td>46</td>
</tr>
<tr>
<td>11.</td>
<td>Kernel Databases</td>
<td>47</td>
</tr>
<tr>
<td>12.</td>
<td>Memory Management Unit (MMU) Image</td>
<td>50</td>
</tr>
<tr>
<td>13.</td>
<td>SASS H/W System Overview</td>
<td>59</td>
</tr>
<tr>
<td>14.</td>
<td>Global Active Segment Table</td>
<td>64</td>
</tr>
<tr>
<td>15.</td>
<td>Alias Table Creation</td>
<td>67</td>
</tr>
<tr>
<td>16.</td>
<td>Local Active Segment Table</td>
<td>69</td>
</tr>
<tr>
<td>17.</td>
<td>Alias Table</td>
<td>70</td>
</tr>
<tr>
<td>18.</td>
<td>Memory Management Unit Image</td>
<td>73</td>
</tr>
<tr>
<td>19.</td>
<td>Memory Allocation/Deallocation Map</td>
<td>75</td>
</tr>
<tr>
<td>20.</td>
<td>Memory Manager Mainline Code</td>
<td>77</td>
</tr>
<tr>
<td>21.</td>
<td>Create Entry Pseudo-Code</td>
<td>79</td>
</tr>
</tbody>
</table>
PART A

INTRODUCTION
Chapter I

BACKGROUND

This chapter is an updated excerpt from *Implementation of Segment Management for a Secure Archival Storage System* by J. T. Wells [20].

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 [7]. 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 [9]. No implementation or further design effort on the supervisor has followed, to date. The initial design of the kernel was completed by Coleman [2]. 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.


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 [5] in conjunction with Reitz [12]. 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. Wells provided the implementation of the Segment Manager and Non-Discretionary Security Modules as well as partial implementation of Distributed Memory Manager functions. Strickler refined and implemented the process management functions for the SASS (written in PLZ/ASM).
Chapter II
BASIC CONCEPTS/DEFINITIONS

This chapter is an excerpt from *Implementation of Process Management for a Secure Archival Storage System* by A. R. Strickler [19]. Minor changes have been made for integration into report.

This section provides an overview of several concepts essential to the SASS design. Readers familiar with SASS or with secure operating system principles may wish to skip to the next section.

A. PROCESS

The notion of a process has been viewed in many ways in computer science literature. Organick [8] defines a process as a set of related procedures and data undergoing execution and manipulation, respectively, by one of possibly several processors of a computer. Madnick and Donovan [6] view a process as the locus of points of a processor executing a collection of programs. Reed [10] describes a process as the sequence of actions taken by some processor. In other words, it is the past, present, and future "history" of the states of the processor. In the SASS design, a process is viewed as a logical entity entirely characterized by an address space and an execution point. A process' address space consists of the set of all memory locations accessible
by the process during its execution. This may be viewed as a set of procedures and data related to the process. The execution point is defined by the state of the processor at any given instant of process execution.

As a logical entity, a process may have logical attributes associated with it, such as a security access class, a unique identifier, and an execution state. This notion of logical attributes should not be confused with the more typical notion of physical attributes, such as location in memory, page size, etc. In SASS, a process is given a security access class, at the time of its creation, to specify what authorization it possesses in terms of information access (to be discussed in the next section). It is also given a unique identifier that provides for its identification by the system and is utilized for interaction among processes. A process may exist in one of three execution states: 1) running, 2) ready, and 3) blocked. In order to execute, a process must be mapped onto (bound to) a physical processor in the system. Such a process is said to be in the "running" state. A process that is not mapped onto a physical processor, but is otherwise ready to execute, is in the "ready" state. A process in the "blocked" state is waiting for some event to occur in the system and cannot continue execution until the event occurs. At that time, the process is placed into the ready state.
There is an ever increasing demand for computer systems that can provide controlled access to the data it stores. In this thesis, "information security" is defined as the process of controlling access to information based upon proper authorization. The critical need for information security should be clear. Banks and other commercial enterprises risk the theft or loss of funds. Insurance and credit companies are bound by law to protect the private or otherwise personal information they maintain on their customers. Universities and scientific institutions must prevent the unauthorized use of their often overburdened systems. The Department of Defense and other government agencies must face the very real possibility that classified information is being compromised or that weapon systems are being tampered with. In fact, security related problems can be found at virtually every level of computer usage.

The security of computer systems processing sensitive information can be achieved by two means: external security controls and internal security controls. In the first case, security is achieved by encapsulating the computer and all its trusted users within a single security perimeter established by physical means (e.g., armed guards, fences, etc.) This means of security is often undesirable due to its added cost of implementation, the inherent risk of error-prone manual procedures, and the problem of trustworthy but error-
prone users. Also, since all security controls are external to the computer system, the computer is incapable of securely handling data at differing security levels or users with differing degrees of authorization. This restriction greatly limits the utility of modern computers. Internal security controls rely upon the computer system to internally distinguish between multiple levels of information classification and user authorization. This is clearly a more desirable and flexible approach to information security. This does not mean, however, that external security is not needed. The optimal approach would be to utilize internal security controls to maintain information security and external security controls to provide physical protection of our system against sabotage, theft, or destruction. The primary concern of this thesis is information security and will therefore center its discussion on the achievement of information security through implementation of the security kernel concept.

One might argue that a "totally secure" computer system is one that allows no access to its classified or otherwise sensitive information. Such a system would not be of much value to its users. Therefore, when we say that a system provides information security, it is only secure with respect to some specific external security policy established by laws, directives, or regulations. There are two distinct aspects of security policy: non-discretionary and discre-
tionary. Each user (subject) of the system is given a label denoting what classification or level of access the user is authorized. Likewise, all information or segments (objects) within the system are labelled with their classification or level of sensitivity. The non-discretionary security mechanism is responsible for comparing the authorization of a subject with the classification of an object and determining what access, if any, should be granted. The DOD security classification system provides an example of the non-discretionary security policy and is the policy implemented in SASS. The discretionary security policy is a refinement of the non-discretionary policy. As such, it adds a higher degree of restriction by allowing a subject to specify or restrict who may have access to his files. It must be emphasized that the discretionary policy is contained within the non-discretionary policy and in no way undermines or substitutes for it. This prevents a subject from granting access that would violate the non-discretionary policy. An example of discretionary security is provided by the DOD "need to know" policy. In SASS, the discretionary policy is implemented within the supervisor [9] by means of an Access Control List (ACL). There is an ACL maintained for every file in the system, which provides a list of all users authorized access to that file. Every attempt by a user to access a file is first checked against the ACL and then checked against the non-discretionary security policy. The "least"
or "most restrictive" access found in these checks is then granted to the user.

The relationship between the labels associated with the subject's access class (sac) and the object's access class (oac) is defined by a lattice model of secure information flow [12] as follows ("I" denotes "no relationship"):
1. \( \text{sac} = \text{oac} \), read and write access permitted
2. \( \text{sac} > \text{oac} \), read access permitted
3. \( \text{sac} < \text{oac} \), write access permitted
4. \( \text{sac} \not< \text{oac} \), no access permitted

In order to understand how these access levels are determined, it is necessary to gain an awareness of and consideration for several basic security properties.

The "Simple Security Property" deals with "read" access. It states that a subject may have read access only to those object's whose classification is less than or equal to the classification of the subject. This prevents a subject from reading any object possessing a classification higher than his own.

The "Confinement Property" (also known as "*-property") governs "write" access. It states that a user may be granted write access only to those objects whose classification is greater than or equal to the classification of the subject. This prevents a user from writing information of a higher classification (e.g., Secret) into a file of a lower classification (e.g., Unclassified).
this property allows a user to write into a file possessing a classification higher than his own, it does not allow his access to any of the data in that file. The SASS design does not allow a user to "write up" to higher classified files. Therefore, in SASS, "sac < oac" denotes "no access permitted."

The "Compatibility Property" deals with the creation of objects in a hierarchical structure. In SASS, objects (segments) are hierarchically organized in a tree structure. This structure consists of nodes with a root node from which the tree emanates. The Compatibility Property states that the classification of objects must be non-decreasing as we move down the hierarchical structure. This prevents a parent node from creating a child node of a lower classification.

Several prerequisites must be met in order to insure that the security kernel design provides a secure environment. Firstly, every attempt to access data must invoke the Kernel. In addition, the Kernel must be isolated and tamperproof. Finally, the Kernel design must be verifiable. This implies that the mathematical model, upon which the Kernel is based, must be proved secure and that the Kernel is shown is to correctly implement this model.
C. SEGMENTATION

Segmentation is a key element of a security Kernel based system. A segment can be defined as a logical grouping of information, such as a procedure, file or data area [6]. Therefore, we can redefine a process' address space as the collection of all segments addressable by that process. Segmentation is the technique applied to effect management of those segments within an address space. In a segmented environment, all references within an address space require two components: 1) a segment specifier (number) and 2) the location (offset) within the segment.

A segment may have several logical and physical attributes associated with it. The logical attributes may include the segment’s classification, size, or permissible access (read, write, or execute). These logical attributes allow a segment to nicely fit the definition of an object within the security kernel concept, and thus provide a means for the enforcement of information security. A segment’s physical attributes include the current location of the segment, whether or not the segment resides in main memory or secondary storage, and where the segment’s attributes are maintained by a segment descriptor. The segment descriptors for each segment in a process’ address space are contained within a Descriptor Segment (viz., the MAN Image in SASS) to facilitate the memory management of that address space.
Segmentation supports information sharing by allowing a single segment to exist in the address spaces of multiple processes. This allows us to forego the maintenance of multiple copies of the same segment and eliminates the possibility of conflicting data. Controlled access to a segment is also enforced, since each process can have different attributes (read/write) specified in its segment descriptor.

In the implementation of SASS, any segment which is shared, but has "read only" access by every process sharing it, is placed in the processor local memory supporting each of these processes rather than in the global memory. This implies the maintenance of multiple copies of some shared segments. It is noted that the problem of "conflicting data" is avoided since this only applies to read only segments. This apparent waste of memory and nonuse of existing sharing facilities is justified by a design decision to provide maximum reduction of bus contention among processors accessing global memory. This reduction in bus contention is considered to be of more importance than the saving of memory space provided by single copy sharing of read only segments. This decision is also well supported by the occurrence of decreasing memory costs, which we have experienced in terms of high speed bus costs.
D. **PROTECTION DOMAINS**

The requirement for isolating the Kernel from the remainder of the system is achieved by dividing the address space of each process into a set of hierarchical domains or protection rings [18]. O’Connell and Richardson [7] defined three domains in the family of secure operating systems: the user, the supervisor, and the kernel. Only two domains are actually necessary in the SASS design since it does not provide extended user applications. The Kernel resides in the inner or most privileged domain and has access to all segments in an address space. System-wide data bases are also maintained within the Kernel domain to insure their accessibility is only through the Kernel. The Supervisor exists in the outer or least privileged domain where its access to data or segments within an address space is restricted.

While protection domains may be created through either hardware or software mechanisms, a hardware implementation provides much greater efficiency. Current microprocessor technology only provides for the implementation of two domains. This two-domain restriction does not support O’Connell and Richardson’s complete family design, but it is sufficient to allow hardware implementation of the ring structure required by the SASS subset.
E. ABSTRACTION

Dijkstra [4] has shown that the notion of abstraction can be used to reduce the complexity of a problem by applying a general solution to a number of specific cases. A structure of increasing levels of abstraction provides a powerful tool for the design of complex systems and generally leads to a better design with greater clarity and fewer errors.

Each level of abstraction creates a virtual hierarchical machine [6] which provides a set of "extended instructions" to the system. A virtual machine cannot make calls to another virtual machine at a higher level of abstraction and in fact is unaware of its existence. This implies that a level of abstraction is independent of any higher levels. This independence provides for a loop-free design. Additionally, a higher level may only make use of the resources of a lower level by applying the extended instruction set of the lower level virtual machine. Therefore, once a level of abstraction is created, any higher level is only interested in the extended instruction set it provides and is not concerned with the details of its implementation. In SASS, once a level of abstraction is created for the physical resources of the system, these resources become "virtualized" making the higher levels of the design independent of the physical configuration of the system.
PART B
SECURE ARCHIVAL STORAGE SYSTEM DESIGN

This section is an excerpt from Implementation of Process Management for a Secure Archival Storage System by A. R. Strickler [19]. Minor changes have been made for integration into this report.
Chapter III

BASIC SASS OVERVIEW

The purpose of the Secure Archival Storage System is to provide a secure "data warehouse" or information pool which can be accessed and shared by a variable set of host computer systems possessing differing security classifications. The primary goals of the SASS design are to provide information security and controlled sharing of data among system users.

Figure 5 provides an example of a possible SASS usage. The system is used exclusively for managing an archival storage system and does not provide any programming services to its users. Thus the users of the SASS may only create, store, retrieve, or modify files within the SASS. The host computers are hardwired to the system via the I/O ports of the Z8001 with each connection having a fixed security classification. Each host must have a separate connection for each security level it wishes to work on (It is important to note that Figure 5 only represents the logical interfacing of the system. Specifically, the actual connection with the host system must be interfaced with the Kernel as the I/O instructions for the port are privileged). In our example, Host #1 can create and modify only Top Secret files, but it
can read files which are Top Secret, Secret, Confidential, or Unclassified. Likewise, Host #2 can create or modify secret files, using its secret connection or confidential files, using its confidential connection. Host #2 cannot create or modify Top Secret or Unclassified files.

In order to provide information security and controlled sharing of files, the SASS operates in two domains: (1) the Supervisor domain and (2) the Kernel domain. The SASS achieves this desired environment through a distributed operating system design which consists of two primary modules: the Supervisor and the Security Kernel. Each host system connected to the SASS has associated with it two processes within the SASS which perform the data transfer and file management on behalf of that host. The host computer communicates directly with its own I/O process and File Manager process within the SASS.

We can use our notion of abstraction to present a system overview of the SASS. The SASS consists of four primary levels of abstraction:

- Level 3-The Host Computer Systems
- Level 2-The Supervisor
- Level 1-The Security Kernel
- Level 0-The SASS Hardware

A pictorial representation of this abstract system overview is presented in Figure 6. This representation is limited to a dual host system for clarity and space restrictions. Note
Figure 5: SASS System
that the Gate Keeper module is in actuality the logical boundary between levels one and two and as such will be described separately.

Level 3, the host computer systems, of SASS has already been addressed. It should be noted that the SASS design makes no assumptions about the host computer systems. Therefore each host may be of a different type or size (i.e.- micro, mini, or maxi-computer system). Furthermore, the necessary physical security of the host systems and their respective data links with the SASS is assumed.
Figure 6: System Overview (Dual Host)
Chapter IV

SUPERVISOR

Level 2 of the SASS system is composed of the Supervisor domain. As already stated, the SASS consists of two domains. The actual implementation of these domains was greatly simplified since the Z8001 microprocessor provides two modes of execution. The system mode, with which the Kernel was implemented, provides access to all machine instructions and all segments within the system. The normal mode, with which the Supervisor was implemented, only provides access to a limited subset of machine instructions and segments within the system. Therefore, the Supervisor operates in an outer or less privileged domain than the Kernel.

The purpose of the Supervisor is to manage the data link between the host computer systems and the SASS by means of Input/Output control, and to create and manage the file hierarchy of each host within the SASS. These functions are accomplished via an Input/Output (I/O) process and a File Manager (FM) process within the Supervisor. A separate FM and I/O process are created and dedicated to each host at the time of system initialization.
A. FILE MANAGER PROCESS

The FM process directs the interaction between the host computer systems and the SASS. It interprets all commands received from the Host computer and performs the necessary action upon them through appropriate calls to the Kernel. The primary functions of the FM process are the management of the Host's virtual file system and the enforcement of the discretionary security policy.

The virtual file system of the Host is viewed as a hierarchy of files which are implemented in a tree structure. The five basic actions which may be initiated upon a file at this level are: 1) to create a file, 2) to delete a file, 3) to read a file, 4) to store a file, and 5) to modify a file. The FM process utilizes a File Known Segment Table (FM_KST) as the primary database to aid in this management.

The FM process maintains an Access Control List (ACL) through which it enforces the discretionary security in SASS. The FM process initializes an ACL for every file in its Host's file system. The ACL is merely a list of all users that are authorized to access that file. The ACL is checked upon every attempt to access a file to determine its authorization. The user (host computer) directs the FM process as to what entries or deletions should be made in the ACL, and as such, specifies who he wishes to have access to his file. As noted earlier, discretionary security is a refinement to the Non-Discretionary Security Policy and there-
fore can only be utilized to add further access restrictions to those provided by the Non-Discretionary Security. This prevents a user from granting access to a file to someone who otherwise would not be authorized access.

B. INPUT/OUTPUT PROCESS

The I/O process is responsible for managing the input and output of all data between the host computer systems and the SASS. The I/O process is subservient to the FM process and receives all of its commands from it. Data is transferred between the SASS and Host Computer systems in fixed size "packets". These packets are broken up into three basic types: 1) a synchronization packet, 2) a command packet, and 3) a data packet. In order to insure reliable transmission and receipt of packets between the Host computer and the SASS, there must exist a protocol between them. Parks [9] provides a more detailed description of these packets, and a possible multi-packet protocol.
Chapter V

GATE KEEPER

The primary objective of the gate keeper is to isolate the Kernel and make it tamperproof. This goal is accomplished by reason of a software ring crossing mechanism provided by the gate keeper. In terms of SASS, this notion of "ring-crossing" is merely the transition from the Supervisor domain to the Kernel domain. As noted earlier, the gate keeper establishes the logical boundary between the Supervisor and the Kernel, and as a matter of course, it provides a single software entry point (enforced by hardware) into the Kernel. Therefore, any call to the Kernel must first pass through the gate keeper.

The gate keeper acts as a trap handler. Once it is invoked by a user (Supervisor) process, the hardware preempt interrupts are masked, and the user process' registers and stack pointer are saved (within the kernel domain). It then takes the argument list provided by the caller and validates these passed parameters to insure their correctness. To aid in the validation of these parameters, the gate keeper utilizes the Parameter Table as a database. The Parameter table contains all of the permitted functions provided by the Kernel. These relate directly to the extended instruction
set (viz., Supervisor calls) provided by the Kernel (these extended instructions will be described in the next section). If an invalid call is encountered by the gate keeper, an error code is returned, and the Kernel is not invoked. If a valid call is encountered by the gate keeper, the arguments and control are passed to the appropriate Kernel module.

Once the Kernel has completed its action on the user request, it passes the necessary parameters and control back to the gate keeper. At this point, the gate keeper determines if any software virtual preempt interrupts have occurred. If they have, then the virtual preempt handler is invoked vice the Kernel being exited (virtual interrupt structure is discussed by Strickler [19]. Correspondingly, if a software virtual preempt has not occurred, then the return arguments are passed to the user process. The user process' registers and stack pointer (viz., its execution point) are restored and control returned to the Supervisor domain. A detailed description of the Gate Keeper interface and implementation is provided by Strickler [19].
Chapter VI

DISTRIBUTED KERNEL

Level 1 of our abstract view of SASS consists of two components: the distributed Kernel and the non-distributed Kernel. These two elements comprise the Security Kernel of the SASS. The Security Kernel has two primary objectives: 1) the management of the system's hardware resources, and 2) the enforcement of the non-discretionary security policy. It executes in the most privileged domain (viz., the system mode of the Z8001) and has access to all machine instructions. The following section will provide a brief description of the distributed Kernel, its components, and the extended instruction set it provides. A discussion of the non-distributed Kernel will be given in the next section.

The distributed Kernel consists of those Kernel modules whose segments are contained (distributed) in the address space of every user (Supervisor) process. Thus, in effect, the distributed Kernel is shared by all user processes in the SASS. The distributed Kernel is composed of the Segment Manager, the Event Manager, the Non-Discretionary Security Module, the Traffic Controller, the Inner Traffic Controller, and the Distributed Memory Manager Module. The Segment Manager and the Event Manager are the only "user visible"
modules in the distributed kernel. In other words, the set of extended instructions available to user processes invokes either the Segment Manager or the Event Manager.

A. SEGMENT MANAGER

The objective of the Segment Manager is the management of a process' segmented virtual storage. The Segment Manager is invoked by calls from the Supervisor domain via the gate keeper. Calls to the Segment Manager are made by means of six extended instructions provided by the segment manager. These extended instructions (viz., entry points) are: 1) CREATE_SEGMENT, 2) DELETE_SEGMENT, 3) MAKEKNOWN, 4) TERMINATE, 5) SM_SWAP_IN, and 6) SM_SWAP_OUT. The extended instructions CREATE_SEGMENT and DELETE_SEGMENT add and remove segments from the SASS. MAKEKNOWN and TERMINATE add and remove segments from the address space of a process. Finally, SM_SWAP_IN and SM_SWAP_OUT move segments from secondary storage to main storage and vice versa.

The primary database utilized by the Segment Manager is the Known Segment Table (KST). A representation of the structure of the KST is provided in Figure 7. The KST is a process local database that contains an entry for every segment in the address space of that process. The KST is indexed by segment number with each record of the KST containing descriptive information for a particular segment. The KST provides a mapping mechanism by which the segment number
of a particular segment can be converted into a unique handle for use by the Memory Manager. The Memory Manager will be discussed in the next chapter.
<table>
<thead>
<tr>
<th>Segment</th>
<th>MM Handle</th>
<th>Size</th>
<th>Access Mode</th>
<th>Xn Core</th>
<th>Class</th>
<th>Mentor</th>
<th>Entry Seg No</th>
<th>Entry Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: Known Segment Table (KST)
B. EVENT MANAGER

The purpose of the Event Manager is the management of event data which is associated with interprocess communications within the SASS. This event data is implemented by means of eventcounts (a synchronization primitive discussed by Reed [11]). The Event Manager is invoked, via the Gate Keeper, by user processes residing in the Supervisor domain. There are two eventcounts associated with every segment existing in the Supervisor domain. These eventcounts (viz., Instance 1 and Instance 2) are maintained in a database residing in the Memory Manager. The Event Manager provides its management functions through its extended instruction set READ, TICKET, ADVANCE, and AWAIT, and in conjunction with the extended instructions TC_ADVANCE and TC_AWAIT provided by the Traffic Controller (to be discussed next). These extended instructions are based on the mechanism of eventcounts and sequencers [11]. The Event Manager verifies the access permission of every interprocess communication request through the Non-Discretionary Security Module. The extended instruction READ provides the current value of the eventcount requested by the caller. TICKET provides a complete time ordering of possibly concurrent events through the mechanism of sequencers. The Event Manager will be discussed in more detail by Strickler [19].

- 32 -
C. NON-DISCRETIONARY SECURITY MODULE

The purpose of the Non-Discretionary Security Module (NDS) is the enforcement of the non-discretionary security policy of the SASS. While the current implementation of SASS represents the Department of Defense security policy, any security policy which may be represented through a lattice structure [3] may also be implemented. The NDS is invoked via its extended instruction set: CLASS_EQ and CLASS_GE. The NDS is passed two classifications which it compares and then analyzes their relationship. CLASS_EQ will return a true value to the calling procedure only if the two classifications passed were equal. The CLASS_GE instruction will return true if a given classification is analyzed to be either greater than or equal to another given classification. The NDS does not utilize a data base as it works only with the parameters it is passed.

D. TRAFFIC CONTROLLER

The task of processor scheduling is performed by the traffic controller. Saltzer [14] defines traffic controller as the processor multiplexing and control communication section of an operating system. The current SASS design utilizes Reed's [10] notion of a two level traffic controller, consisting of: 1) a Traffic Controller (TC) and 2) an Inner Traffic Controller (ITC).

- 33 -
The primary function of the Traffic Controller is the scheduling (binding) of user processes onto virtual processors. A virtual processor (VP) is an abstract data structure that simulates a physical processor through the preservation of an executing process' attributes (viz., the execution point and address space). Multiple VP's may exist for every physical processor in the system. Two VP's are permanently bound to Kernel processes (viz., Memory Manager and Idle) and as such are not in contention for process scheduling. These processes and their corresponding virtual processors are invisible to the TC. The remaining virtual processors are either idle or are temporarily bound to user processes as scheduled by the TC. The database utilized by the TC in process scheduling is the Active Process Table (APT). Figure 8 provides the structure of the APT.

The APT is a system-wide Kernel database containing an entry for every user process in the system. Since the current SASS design does not provide for dynamic process creation/deletion, a user process is active for the life of the system. Therefore, the size of the APT is fixed at the time of system generation. The APT is logically composed of three parts: 1) an APT header, 2) the main body of the APT, and 3) a VP table. The APT header includes: 1) a Lock to provide for a mutual exclusion mechanism, 2) a Running List indexed by VP ID to identify the current process running on each VP, 3) a Ready List, which points to the linked list of
Figure 8: Active Process Table (APT)
processes which are ready for scheduling, and 4) a Blocked List, which points to the linked list of processes which are in the blocked state awaiting the occurrence of some event.

A design decision was made to incorporate a single list of blocked processes instead of the more traditional notion of separate lists per event count because of its simplicity and its ease of implementation. This decision does not appreciably affect system performance or efficiency as the "blocked" list will never be very long. The VP table is indexed by logical CPU number and specifies the number of VP's associated with the logical CPU and its first VP in the Running List. The logical CPU number, obtained during system initialization, provides a simple means of uniquely identifying each physical CPU in the system. The main body of the APT contains the user process data required for its efficient control and scheduling. NEXT_AP provides the linked list threading mechanism for process entries. The DBR entry is a handle identifying the process' Descriptor Segment which is employed in process switching and memory management. The ACCESS_CLASS entry provides every process with a security label that is utilized by the Event Manager and the Segment Manager in the enforcement of the Non-Discretionary Security Policy. The PRIORITY and STATE entries are the primary data used by the Traffic Controller to effect process scheduling. AFFINITY identifies the logical CPU which is associated with the process. VP ID is utilized to iden-
tify the virtual processor that is currently bound to the process. Finally, the EVENTCOUNT entries are utilized by the TC to manage processes which are blocked and awaiting the occurrence of some event. HANDLE identifies the segment associated with the event, INSTANCE specifies the event, and COUNT determines which occurrence of the event is needed.

The Traffic Controller determines the scheduling order by process priority. Every process is assigned a priority at the time of its creation. Once scheduled, a process will run on its VP until it either blocks itself or it is preempted by a higher priority process. To insure that the TC will always have a process available for scheduling, there logically exists an "idle" process for every VP visible to the TC. These "idle" processes exist at the lowest process priority and, consequently, are scheduled only if there exists no useful work to be performed.

The Traffic Controller is invoked by the occurrence of a virtual preempt interrupt or through its extended instruction set: ADVANCE, AWAIT, PROCESS_CLASS, and GET,DBR_NUMBER. ADVANCE and AWAIT are used to implement the IPC mechanism invoked by the Supervisor. PROCESS_CLASS and GET,DBR_NUMBER are called by the Segment Manager to ascertain the security label and DBR handle, respectively, of a named process. A more detailed discussion of the TC is provided by Strickler [19].
E. INNER TRAFFIC CONTROLLER

The Inner Traffic Controller is the second part of our two-level traffic controller. Basically, the ITC performs two functions. It multiplexes virtual processors onto the actual physical processors, and it provides the primitives for which inter-VP communication within the Kernel is implemented. A design choice was made to provide each physical processor in the system with a small fixed set of virtual processors. Two of these VP's are permanently bound to the Kernel processes. The Memory Manager is bound to the highest priority VP. Conversely, the Idle Process is bound to the lowest priority VP and, as a result, will only be scheduled if there exists no useful work for the CPU to perform.

The primary database utilized by the ITC is the Virtual Processor Table (VPT). Figure 9 illustrates the VPT.

The VPT is a system wide Kernel database containing entries for every CPU in the system. The VPT is logically composed of four parts: 1) a header, 2) a VP data table, 3) a message table, and 4) an external VP list. The header includes a LOCK (spin lock) that provides a mutual exclusion mechanism for table access, a RUNNING LIST (indexed by logical CPU #) that identifies the VP currently running on the corresponding physical CPU, a READY LIST (indexed by logical CPU #) which points to the linked list of VP's which are in the "ready" state and awaiting scheduling on that CPU, and a FREE LIST which points to the linked list of unused entries.
Figure 9: Virtual Processor Table (VPT)
in the message table. The VP data table contains the descriptive data required by the ITC to effectively manage the virtual processors. The DBR entry points within the MAU Image to the descriptor segment for the process currently running on the VP. PRI (Priority), STATE, IDLE_FLAG, and PREEMPT are the primary data used by the ITC for VP scheduling. PREEMPT indicates whether or not a virtual preempt is pending for the VP. The IDLE_FLAG is set whenever the TC has bound an "idle" process to the VP. Normally, a VP with the IDLE_FLAG set will not be scheduled by the ITC as it has no useful work to perform. In fact, such a VP will only be scheduled if the PREEMPT flag is set. This scheduling will allow the VP to be given (bound) to another process. PHYSICAL PROCESSOR contains an entry from the Processor Data Segment (PRDS) that identifies the physical processor that the VP is executing on. EXT_VP_ID is the identifier by which the VP is known by the Traffic Controller. A design choice was made to have the EXT_VP_ID equate to an offset into the External VP List. The External VP List specifies the actual VP ID (viz., VPT entry number) for each external VP identifier. This precluded the necessity for run time calculation of offsets for the EXT_VP_ID. NEXT_READY_VP provides the threading mechanism for the "Ready" linked list, and MSG_LIST points to the first entry in the Message Table containing a message for that VP. The Message Table provides storage for the messages generated in the course of
Inter-Virtual Processor communications. MSG contains the actual communication being passed, while SENDER identifies the VP which initiated the communication. NEXT_MSG provides a threading mechanism for multiple messages pending for a single VP.

The ITC is invoked by means of its extended instruction set: WAIT, SIGNAL, SWAP_VDBR, IDLE, SETPRETTY, and RUNNING_VP. WAIT and SIGNAL are the primitives employed in implementing the Inter-VP communication. SWAP_VDBR, IDLE, SETPRETTY, and RUNNING_VP are all invoked by the Traffic Controller. SWAP_VDBR provides the means by which a user process is temporarily bound to a virtual processor. IDLE binds the "Idle" process to a VP (the implication of this instruction will be discussed later). SETPRETTY provides the means of indicating that a virtual preempt interrupt is pending on a VP (specified by the TC) by setting the PREEMPT flag for that VP in the VPT. RUNNING_VP provides the TC with the external VP ID of the virtual processor currently running on the physical processor.

F. DISTRIBUTED MEMORY MANAGER

The Distributed Memory Manager provides an interface structure between the Segment Manager and the Memory Manager Process. This interfacing is necessitated by the fact that the Memory Manager Process does not reside in the Distributed Kernel and consequently is not included in the user pro-
cess' address space. The primary functions performed in this module are the establishment of Inter-VP Communication between the VP bound to its user process and the VP permanently bound to the Memory Manager Process, the manipulation of event data, and the dynamic allocation of available memory. The Distributed Memory Manager Module is invoked by the Segment Manager through its extended instruction set: MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. These extended instructions are utilized on a one-to-one basis by the extended instruction set of the Segment Manager (e.g., SM_SWAP_IN utilizes (calls) MM_SWAP_IN). Wells [20] provides a more detailed description of this portion of the Distributed Memory Manager and the extended instruction set associated with it.

The Distributed Memory Manager is also invoked through its remaining extended instructions: MM_READ_EVENTCOUNT, MM_TICKET, MM_ADVANCE, and MM_ALLOCATE. These Distributed Memory Manager functions are discussed in detail by Stricker [19].
Chapter VII
NON-DISTRIBUTED KERNEL

The Non-Distributed Kernel is the second element residing in Level 1 of our abstract system view of the SASS. The sole component of the Non-Distributed Kernel is the Memory Manager Process.

A. MEMORY MANAGER PROCESS

The primary purpose of the Memory Manager Process is the management of all memory resources within the SASS. These include the local and global main memories, as well as the hard-disk based secondary storage. A dedicated Memory Manager Process exists for every CPU in the system. Each CPU possesses a local memory where process local segments and shared, non-writeable segments are stored. There is also a global memory, to which every CPU has access, where the shared, writeable segments are stored. It is necessary to store these shared, writeable segments in the global memory to ensure that a current copy exists for every access.

The Memory Manager Process is tasked by other processes within the Kernel domain (via Signal and Wait) to perform memory management functions. These basic functions include the allocation/deallocation of local and global memory and
of secondary storage, and the transfer of segments between the local and global memory and between secondary storage and the main memories. The extended instruction set provided by the Memory Manager Process includes: CREATE_ENTRY, DELETE_ENTRY, ACTIVATE, DEACTIVATE, SWAP_IN, and SWAP_OUT. These instructions correspond one to one with those of the Distributed Memory Manager Module. The system wide databases utilized by all Memory Manager Processes are the Global Active Segment Table (G_AST), the Alias Table, the Disk Bit Map, and the Global Memory Bit Map. The processor local databases used by each Memory Manager Process are the Local Active Segment Table (L_AST), and the Local Memory Bit Map. Gary and Moore [5] provide a detailed description of the Memory Manager, its extended instruction set, and its databases.

A summary of the extended instruction set created by the components of the Security Kernel is provided by Figure 10. One might question the prudence of omitting PHYS_PREEMPT_HANDLER and VIRT_PREEMPT_HANDLER (viz., the handler routines for physical and virtual interrupts) from the extended instruction set as both of these interrupts may be raised (viz., initiated) from within the Kernel. A decision was made to not classify these handlers as "extended instructions" since they are only executed as the result of a physical or virtual interrupt and as such cannot be directly invoked (viz., "called") by any module in the system.
A summary of the databases utilized by kernel modules is presented in Figure 11.
<table>
<thead>
<tr>
<th>MODULE</th>
<th>INSTRUCTION SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Manager</td>
<td>Create_Segment* Delete_Segment*</td>
</tr>
<tr>
<td></td>
<td>Make_Known*</td>
</tr>
<tr>
<td></td>
<td>SM_Swap_In*</td>
</tr>
<tr>
<td>Event Manager</td>
<td>Read* Ticket*</td>
</tr>
<tr>
<td></td>
<td>Advance* Await*</td>
</tr>
<tr>
<td>Non-Discretionary</td>
<td>Class_EQ Class_GE</td>
</tr>
<tr>
<td>Security</td>
<td></td>
</tr>
<tr>
<td>Traffic Controller</td>
<td>TC_Advance TC_Await</td>
</tr>
<tr>
<td></td>
<td>Process_Class</td>
</tr>
<tr>
<td>Inner Traffic</td>
<td>Signal Wait</td>
</tr>
<tr>
<td>Controller</td>
<td>Swap_VDBR Idle</td>
</tr>
<tr>
<td></td>
<td>Set_Preempt Test_Preempt</td>
</tr>
<tr>
<td></td>
<td>Running_VP</td>
</tr>
<tr>
<td>Distributed Memory</td>
<td>MM_Create_Entry MM_Delete_Entry</td>
</tr>
<tr>
<td>Manager</td>
<td>MM_Activate MM_Deactivate</td>
</tr>
<tr>
<td></td>
<td>MM_Swap_In MM_Swap_Out</td>
</tr>
<tr>
<td>Non-Distributed Memory</td>
<td>Create_Entry Delete_Entry</td>
</tr>
<tr>
<td>Manager</td>
<td>Activate Deactivate</td>
</tr>
<tr>
<td></td>
<td>Swap_In Swap_Out</td>
</tr>
</tbody>
</table>

* Denotes user visible instructions

Figure 10: Extended Instruction Set

- 46 -
<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 (APT)</td>
</tr>
<tr>
<td>Inner Traffic Controller</td>
<td>Virtual_Processor_Table (VPT)</td>
</tr>
<tr>
<td>Memory Manager</td>
<td>Memory_Management_Unit Image (MMU)</td>
</tr>
<tr>
<td></td>
<td>Global_Active_Segment_Table (G_AST)</td>
</tr>
<tr>
<td></td>
<td>Local_Active_Segment_Table (L_AST)</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 11: Kernel Databases

- 47 -
Chapter VIII

SYSTEM HARDWARE

Level 0 of the SASS consists of the system hardware. This hardware includes: 1) the CPU, 2) the local memory, 3) the global memory, 4) the secondary storage (viz. hard disk), and 5) the I/O ports connecting the host computer systems to the SASS. Since the SASS design allows for a multiprocessor environment, there may exist multiple CPU's and local memories. The target machine selected for the initial implementation of the system is the Zilog Z8001 microprocessor [22]. The Z8001 is a general purpose 16-bit, register oriented machine that has sixteen 16-bit general purpose registers. It can directly address 8M bytes of memory, extensible to 48M bytes. The Z8001 architecture supports memory segmentation and two-domain operations. The memory segmentation capability is provided externally by the Zilog Z8010 Memory Management Unit (MMU). The Z8010 MMU [23] provides management of the Z8001 addressable memory, dynamic segment relocation, and memory protection. Memory segments are variable in size from 256 bytes to 64K bytes and are identified by a set of 64 Segment Descriptor Registers, which supply the information needed to map logical memory addresses to physical memory addresses. Each of the 64
Descriptor Registers contains a 16-bit base address field, an 8-bit limit field, and an 8-bit attribute field. Unfortunately, the Z8001 hardware was not available for use during system development. Therefore, all work to date has been completed through utilization of the Z8002 non-segmented version of the Z8000 microprocessor family [22]. The actual hardware used in this implementation is the Advanced Micro Computers Am96/4116 MonoBoard Computer [1] containing the AmZ8002 sixteen bit non-segmented microprocessor. This computer provides 32K bytes of on-board RAM, 8k bytes of PROM/ROM space, two RS232 serial I/O ports, 24 parallel I/O lines, and a standard INTEL Multibus interface. The general structure of the design has been preserved by simulation of the segmentation hardware in software. This software MHU Image (see Figure 12) is created as a database within the Inner Traffic Controller.

The MHU Image is a processor-local database indexed by DBR_No. Each DBR_No represents one record within the MHU Image. Each record is an exact software copy of the Segment Descriptor Register set in the hardware MHU. Each element of this software MHU Image is in the same form utilized by the special I/O instructions to load the hardware MHU. Each DBR record is indexed by segment number (Segment_No). Each Segment_No entry is composed of three fields: Base_Addr, Limit, and Attributes. Base_Addr is a 16-bit field which contains the base address of the segment in physical memory.
<table>
<thead>
<tr>
<th>Segment No.</th>
<th>Base_Addr</th>
<th>Limit</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(entries for one DBR #)

Figure 12: Memory Management Unit (MMU) Image
Limit is an 8-bit field that specifies the number of contiguous blocks of memory occupied by the segment. Attributes is an 8-bit field representing the eight flags which specify the segment's attributes (e.g., "read", "execute", "write", etc.).
Chapter IX

SUMMARY

An extended overview of the current SASS design has been presented. The four major levels of abstraction comprising the SASS system have been identified, and the major components of each level have been discussed. The extended instruction set provided by the SASS Kernel was also defined. The actual details of this implementation are described by Strickler [19].
PART C
THE DESIGN AND IMPLEMENTATION OF THE MEMORY MANAGER FOR A SECURE ARCHIVAL STORAGE SYSTEM

This section contains updated excerpts from a Naval Postgraduate School MS Thesis by E. E. Moore and A. V. Gary [5]. The origins of these excerpts are:

- **INTRODUCTION** from Chapter I
- **MEMORY MANAGER PROCESS DETAILED DESIGN** from Chapter III
- **STATUS OF RESEARCH** from Chapter IV

Minor changes have been made for integration into this report.
THE NAVAL POSTGRADUATE SCHOOL SECURE ARCHIVAL STORAGE SYSTEM. P—ETC(U)
MAR 81 L A COX, R R SCHELL, S L PERDUE
UNCLASSIFIED NPS52-81-001
Chapter I
INTRODUCTION

This thesis addresses the design and partial implementation of a memory manager for a member of the family of secure, distributed, multi-microprocessor operating systems designed by Richardson and O'Connell [7]. The memory manager is responsible for the secure management of the main memory and secondary storage. The memory manager design was approached and conducted with distributed processing, multiprocessing, configuration independence, ease of change, and internal computer security as primary goals. The problems faced in the design were:

1) Developing a process which would securely manage files in a multi-processor environment.
2) Ensuring that if secondary storage was inadvertently damaged, it could usually be recreated.
3) Minimizing secondary storage accesses.
4) Proper parameter passing during interprocess communication.
5) Developing a process with a loop-free structure which is configuration independent.
6) Designing databases which optimize the memory management functions.

The proper design and implementation of a memory management process is vital because it serves as the interface between the physical storage of files in a storage system and the logical hierarchical file structure as viewed by the user (viz., the file system supervisor design by Parks [9]). If the memory manager process does not function properly, the security of that system cannot be guaranteed.

The secure family of operating systems designed by Richardson and O'Connell is composed of two primary modules, the supervisor and the security kernel. A subset of that system was utilized in the design of the Secure Archival Storage System (SASS). The design of the SASS supervisor was addressed by Parks [9], while the security kernel was addressed concurrently by Coleman [2]. The SASS security kernel design is composed of two parts, the distributed kernel and the non-distributed kernel. The design of the distributed kernel was conducted by Coleman [2], and processor management was implemented by Reitz [12]. This thesis presents the design and implementation of the non-distributed kernel. In the SASS design, the non-distributed kernel consists solely of the memory manager.

The design of the memory manager and its data bases was completed. The initial code was written in PLZ/SYS, but could not be compiled due to the lack of a PLZ/SYS compiler.
A thread of the high level code was selected, hand compiled into PLZ/ASM, and run on the Z8000 developmental module.
Chapter XI
MEMORY MANAGER PROCESS DETAILED DESIGN

A. INTRODUCTION

The memory manager is responsible for the management of both main memory (local and global) and secondary storage. It is a non-distributed portion of the kernel with one memory manager process existing per physical processor. The memory manager is tasked (via signal and wait) to perform memory management functions on behalf of other processes in the system. The major tasks of the memory manager are: 1) the allocation and deallocation of secondary storage, 2) the allocation and deallocation of global and local memory, 3) segment transfer from local to global memory (and vice versa), and 4) segment transfer from secondary storage to main memory (and vice versa). There are ten service calls (via signal) which task the memory manager process to perform these functions. The ten service calls are:

CREATE_ENTRY
DELETE_ENTRY
ACTIVATE
DEACTIVATE
SWAP_IN
SWAP_OUT
DEACTIVATE_ALL*
MOVE_TO_GLOBAL*
MOVE_TO_LOCAL*
UPDATE*
Upon completion of the service request, the memory manager returns the results of the operation to the waiting process (via signal). It then blocks itself until it is tasked to perform another service. The hardware configuration managed by the memory manager process is depicted in Figure 13. The shared data bases used by all memory manager processes are the Global Active Segment Table (G_AST), the Alias Table, the Disk Bit Map, and the Global Memory Bit Map. The processor local data bases used by each process are the Local Active Segment Table (L_AST), the Memory Management Unit Images and the Local Memory Bit Map.

* In the current state these service calls are not implemented; therefore, there are currently six service calls.
Figure 13: SASS H/W System Overview

- 59 -
B. DESIGN PARAMETERS AND DECISIONS

Several factors were identified during the design of the memory manager process that refined the initial kernel design of Coleman [2]. The two areas that were modified, were the management of the MMU images and the management of core memory. Both of these functions were managed outside of the memory manager in the initial design. The inclusion of these functions in the memory manager process significantly improved the logical structure of the overall system design. Additional design parameters were established to facilitate the initial implementation. These design parameters need to be addressed before the detailed design of the memory manager process is presented.

It was decided to make the block/page size of both main memory and secondary storage equal in size. This was to simplify the mapping algorithm from secondary storage to main memory (and vice versa). In the initial design the block/page size was set to 512 bytes.

The size of the page table for a segment was set at one page (non-paged page table). This was to simplify implementation, and had a direct bearing on the maximum segment size supported in the memory manager. For example, a page size of 256 bytes will address a maximum segment size of 32,768 bytes, while a page size of 512 bytes will address a segment size of 131,072 bytes.
The size of the alias table was set to one page (non-paged alias table). The number of entries that the alias table will support is limited by the size of the page table (viz., a page size of 512 bytes will support up to 42 entries in the Alias Table).

In the original design, the main memory allocation was external to the memory manager. This was due to the partitioned memory management scheme outlined by Parks [9] and Coleman [2]. In the current design, all address assignment and segment transfer are managed by the memory manager. This design choice enhanced the generality of the design, and provided support for any memory management scheme (either in the memory manager or at a higher level of abstraction). However, the current design still has a maximum core constraint for each process.

Dynamic memory management is not implemented in this design. Each process is allocated a fixed size of physical core. However, it is not a linear allocation of physical memory. The design supports the maximum sharing of segments in local and global memory. All segments that are not shared, or shared and do not violate the readers/writers problem will reside in local memory to eliminate the global bus contention. The need to compact the memory (because of fragmentation) should be minimal in this design due to the maximum sharing of segments. If contiguous memory is not available, the memory manager will compact main memory. After compaction, the memory can be allocated.
The design decision to represent memory as one contiguous block (not partitioned) was made to support a dynamic memory management scheme. Without dynamic memory management, the process' total physical memory cannot exceed the system's main memory. The supervisor knows the size of the segments and the size of the process' virtual core, therefore it can manage the swap in and swap out to ensure that the process' virtual core has not been exceeded.

In the original design, the user's process inner-traffic controller maintained the software images of the memory management unit. This design required the memory manager to return the appropriate memory management data (viz., segment location) to the kernel of the user's process. In the current design, the software images of the MMU are maintained by the memory manager. A descriptor base pointer is provided for the inner-traffic controller to multiplex the process address spaces. The MMU image data base does not need to be locked (to prevent race conditions) due to the fact that process interrupts are masked in the kernel. Thus, if the memory manager (a kernel process) is running then no other process can access the MMU image.

The system initialization process has not been addressed to date. However, this design has made some assumptions about the initial state of the system. Since the memory manager handles the transfer of segments from secondary storage to main memory, it is likely to be one of the first
processes created. The memory manager's core image will consist of its pure code and data sections. The minimal initialization of the memory manager's data bases are entries for the system root and the supervisor's segments in the G_AST and L_AST(s), and the initialization of the MMU images with the kernel segments. The current design does not call for an entry in the G_AST or L_AST for the kernel segments. However, when system generation is designed this will have to be readdressed.

The original [2] memory manager databases have been refined by this thesis to facilitate the memory management functions. The major refinements of the global and local active segment tables are outlined in the following section.

C. **DATA BASES**

1. **Global Active Segment Table**

The Global Active Segment Table (see Figure 14) is a system wide, shared data base used by memory manager processes to manage all active segments. A lock/unlock mechanism is utilized to prevent any race conditions from occurring. The signalling process locks the G_AST before it signals the memory manager. This is done to prevent a deadly embrace from occurring between memory manager processes, and also to simplify synchronization between memory managers. The entire G_AST is locked in this design to simplify the implementation (vice locking each individual entry).
Index #

<table>
<thead>
<tr>
<th>Unique Global Processors</th>
<th>Flag Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Addr</td>
</tr>
<tr>
<td>#0</td>
<td>#1</td>
</tr>
</tbody>
</table>

* Field indicates a two processor environment

<table>
<thead>
<tr>
<th># Active</th>
<th>No.</th>
<th>Page</th>
<th>Alias</th>
<th>Seq-</th>
<th>Inst-</th>
<th>Inst-</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Memory</td>
<td>Active</td>
<td>Size</td>
<td>Table</td>
<td>Table</td>
<td>uencer</td>
<td>ance1</td>
</tr>
<tr>
<td>Depend.</td>
<td>Loc</td>
<td>Loc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14: Global Active Segment Table
The G_AST size is fixed at compile time. The size of the G_AST is the product of the G_AST record size, the maximum number of processes and the number of authorized known segments per process. Although the G_AST is of fixed size, it is plausible to dynamically manage the entries as proposed by Richardson and O'Connell [7]. The current memory manager design could be extended to include this dynamic management.

The Unique_Id field is a unique segment identification number in the G_AST. This field is four bytes wide and will provide over four billion identification numbers. A design choice was made not to manage the reallocation of the unique_id's. Thus when a segment is deleted from the system, the unique_id is not reused.

The Global_Address field is used to indicate if a segment resides in global or local memory. If not null, it contains the global memory base address of a segment. A null entry indicates that the segment might be in local memory(s).

The Processors_L_ASTE_# field is used as a connected processors list. The field is an array structure, indexed by Processor_Id. It identifies which L_AST the segment is active in, and provides the index into each of these tables. The design choice of maintaining an entry in the L_AST for all locally active segments implies that if all entries in the Processors_L_ASTE_# field are null, the segment is not
active and can be removed from the G_AST (viz., no processors are connected).

The Flag_Bits field consists of the written bit, and the writable bit. The written bit is set when a segment is swapped out of memory, and the MMU image indicates that it has been written into. The writable bit is set during segment loading to indicate that some process has write access to that segment.

If an active segment is a leaf, the G_ASTE_#_Parent field provides a back pointer to the G_AST index of its parent. This back pointer to the parent is important during the creation of a segment. If a request is received to create a segment which has a leaf segment as its parent, then an alias table has to be created for that parent. Also, the alias table of the parent’s parent needs to be updated to reflect the existence of the newly created alias table (see Figure 15). The indirect pointer shown is the back pointer to the parent via the G_AST.

The No_Active_In_Memory field is a count of the number of processes that have the segment in global memory. It is used during swap out to determine if the segment can be removed from global memory.

The No_Active_Dependents field is a count of the number of active leaf segments that are dependent on this entry (viz., require that this segment remain in the G_AST). Each time a process activates or deactivates a dependent segment this field is incremented or decremented.
Figure 15: Alias Table Creation
The Size field is the size of the segment in bytes. The Page_Table_location field is the disk location of the page table for a segment, and the Alias_Table_Location field is the disk location of the alias table for the segment. The Alias_Table field can be null to indicate that no alias table exists for the segment.

The last three fields are used in the management of eventcounts and sequencers [12]. The Sequencer field is used to issue a service number for a segment. The Instance_1 field and Instance_2 field are eventcounts (i.e., are used to indicate the next number of occurrences of some event).

2. Local Active Segment Table

The Local Active Segment Table (see Figure 16) is a processor local data base. The L_AST contains the characteristics (viz., segment number, access) of each locally active segment. An entry exists for each segment that is active in a process "loaded" on this CPU and in local memory. The first field of the L_AST contains the memory address of the segment. If the segment is not in memory, this field is used to indicate whether the L_AST entry is available or active. The Segment_No/Access field is a combination of segment number and authorized access. It is an array of records data structure that is indexed by DBR_. The first record element (viz., most significant bit) is used to indicate the access (read or read/write) permitted to that segment. The
second record element (viz., the next seven bits) is used to indicate the segment number. A null segment number indicates that the process does not have the segment active.

Index

<table>
<thead>
<tr>
<th>Memory Addr</th>
<th>Segment #/Access Auth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DBR_0</td>
</tr>
</tbody>
</table>

Figure 16: Local Active Segment Table

3. **Alias Table**

The alias table (see Figure 17) is a memory manager database which is associated with each non-leaf segment in the kernel. An aliasing scheme is used to prevent passing systemwide information (unique_id.) out of the kernel. Segments can only be created through a mentor segment and entry
number into the mentor's alias table. When a segment is created, an entry must be made in its mentor segment's alias table. Thus the mentor segment must be known before that segment can be created.

<table>
<thead>
<tr>
<th>Entry #</th>
<th>Unique_ID</th>
<th>Size</th>
<th>Class</th>
<th>Page Table</th>
<th>Alias Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 17: Alias Table

The alias table consists of a header and an array structure of entries. The header has two "pointers" (viz., disk addresses), one that links the alias table to its associated segment and one that links the alias table to the mentor segment's alias table. The header is provided to support the re-construction of the file system after a system crash due
to device I/O errors. It is not used at all during normal operations. Each entry in the array structure consists of five fields for identifying the created segments. The Unique_Id field contains the unique identification number for the segment. The Size field is used to record the size of the segment. The Class field contains the appropriate security access class of the segment. The Page_Table_Location field has the disk address of the page table. A null entry indicates a zero-length segment. The Alias_Table_Location field has the disk address of the alias table for the segment. A null entry indicates that the segment is a leaf segment.

4. Memory Management Unit Image

The Memory Management Unit Image (MMU_Image) is a processor local data base. It is an array structure that is indexed by the DBR_id. Each MMU_Image (see Figure 18) includes a software representation of the segment descriptor registers (SDR) for the hardware MMU [23]. This is in exactly the format used by the special I/O instructions for loading/unloading the MMU hardware. The SDR contains the Base_Address, Limit and Attribute fields for each loaded segment in the process' address space. The Base_Address field contains the base address of the segments in memory (local or global). The Limit field is the number of blocks of contiguous storage for each segment (zero indicates one
block). The Attribute field contains eight flags. Five flags are used for protecting the segment against certain types of access, two encode the type of accesses made to the segment (read/write), and one indicates the special structure of the segment [23]. Five of the eight flags in the attribute field are used by the memory manager. The "system only" and "execute only" flags are used to protect the code of the kernel from malicious or unintentional modifications. The "read only" flag is used to control the read or write access to a segment. The "change" flag is used to indicate that the segment has been written into, and the "CPU-inhibit" flag is used to indicate that the segment is not in memory.

The last two fields of the MMU Image are the Block_Used field and the Maximum_Available_Blocks field. These two fields are used in the management of each process' virtual core and are not associated with the hardware MMU.
Figure 18: Memory Management Unit Image
5. Memory Allocation/Deallocation Bit Maps

All of the memory allocation/deallocation bit maps (see Figure 19) are basically the same structure. Secondary storage, global memory and local memory are managed by memory bit maps. The Disk_Bit_Map is a global resource that is protected from race conditions via the locking convention for the G_AST. Each bit in the bit map is associated with a block of secondary storage. A zero indicates a free block of storage while a one indicates an allocated block of storage. The Global_Memory_Bit_Map is used to manage global memory. It is a shared resource that is protected from race conditions by the locking of the G_AST. The Local_Memory_Bit_Map is the same structure as the Global_Memory_Bit_Map and is used to manage local memory. The Local_Memory_Bit_Map is not locked since it is not a shared resource between memory managers.
Memory Bit Map

Page 0 1 2 3 4 5 6 7 8 9 1 1 1 1 1 1 1 .... 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5
0 1 2 3 4 7 8 9 0 1 2 3 4 5

Figure 19: Memory Allocation/Deallocation Map

D. BASIC FUNCTIONS

The detailed source code for the basic functions and main line of the memory manager is presented in Appendix J.

In the discussion of the memory manager design, a pseudo-code similar to PLZ/SYS is utilized. The rationale for using this pseudo-code was to provide a summary of the memory manager source code, and to facilitate the presentation of this design.

It is assumed that the memory manager is initialized into the ready state at system generation (as previously mentioned). When the memory manager is initially placed into the running state, it will block itself (via a call to
the kernel primitive Wait). Wait will return a message from a signalling process. This message is interpreted by the memory manager to determine the requested function and its required arguments. The function code is used to enter a case statement, which directs the request to the appropriate memory manager procedure.

When the requested action is completed, the memory manager returns a success code (and any additional required data) to the signalling process via a call to the kernel primitive Signal. This call will awaken the process which requested the action to be taken, and place the returned message into that process' message queue. When that action is completed, the memory manager will return to the top of the loop structure and block itself to wait for the next request. The main line pseudo-code of the memory manager process is displayed in Figure 20.
ENTRY

INITIALIZE_PROCESSOR_LOCAL_VARIABLES

DO

! CHECK_IF_MSG_QUEUE_EMPTY!

VP_ID, MSG := WAIT

FUNCTION, ARGUMENTS := VALIDATE_MSG(MSG)

IF FUNCTION

CASE CREATE_ENTRY THEN
SUCCESS_CODE := CREATE_ENTRY(ARGUMENTS)
CASE DELETE_ENTRY THEN
SUCCESS_CODE := DELETE_ENTRY(ARGUMENTS)
CASE ACTIVATE THEN
SUCCESS_CODE := ACTIVATE(ARGUMENTS)
CASE DEACTIVATE THEN
SUCCESS_CODE := DEACTIVATE(ARGUMENTS)
CASE SWAP_IN THEN
SUCCESS_CODE := SWAP_IN(ARGUMENTS)
CASE SWAP_OUT THEN
SUCCESS_CODE := SWAP_OUT(ARGUMENTS)
CASE DEACTIVATE_ALL THEN
SUCCESS_CODE := DEACTIVATE_ALL(ARGUMENTS)
CASE MOVE_TO_GLOBAL THEN
SUCCESS_CODE := MOVE_TO_GLOBAL(ARGUMENTS)
CASE MOVE_TO_LOCAL THEN
SUCCESS_CODE := MOVE_TO_LOCAL(ARGUMENTS)
CASE UPDATE THEN
SUCCESS_CODE := UPDATE(ARGUMENTS)

FI

SIGNAL(VP_ID, SUCCESS_CODE, ARGUMENTS)

OD

END MEMORY_MANAGER_PLZ/SYS MODULE

Figure 20: Memory Manager Mainline Code
1. **Create an Alias Table Entry**

Create_Entry is invoked when a user desires to create a segment. A segment is created by allocating secondary storage, and by making an entry (unique id, secondary storage location, size, classification) into its mentor segment's alias table. This implies that the mentor segment must have an alias table associated with it, and that the mentor segment must be active in order to obtain the secondary storage location of the alias table.

The mentor segment can be in one of two states. It may have children (viz., have an alias table), or it may be a leaf segment (viz., not have an alias table). If the mentor segment has children, it has an alias table and this alias table can be read into core, secondary storage can be allocated, and the data can be entered into the alias table. If the mentor segment is a leaf, an alias table must be created for that segment before it (the alias table) can be read into core and data entered into it (see Figure 15).

The pseudo-code for CREATE_ENTRY PROCEDURE is presented in Figure 21. The arguments passed to Create_Entry are the index into the G_AST for the mentor segment, the entry number into its alias table, the size of the segment to be created, and the security access class of that segment. The return parameter is a success code, which would be "seg_created" for a successful segment creation.
CREATE_ENTRY PROCEDURE (PAR_INDEX WORD, ENTRY_# WORD, SIZE WORD, CLASS BYTE) RETURNS (SUCCESS_CODE BYTE) LOCAL BLKS WORD, PAGE_TABLE_LOC WORD ENTRY IF ALIAS_TABLE_DOES_NOT_EXIST THEN SUCCESS_CODE := CREATE_ALIAS_TABLE IF SUCCESS_CODE <> VALID THEN RETURN FI BLKS := CALCULATE_NO_BLKS_REQ (SIZE) SUCCESS_CODE := READ_ALIAS_TABLE (G_AST[PAR_INDEX].ALIAS_TABLE_LOC) IF SUCCESS_CODE <> VALID THEN RETURN FI SUCCESS_CODE := CHECK_DUP_ENTRY ! in alias table ! IF SUCCESS_CODE <> VALID THEN RETURN FI SUCCESS_CODE, PAGE_TABLE_LOC := ALLOC_SEC_STORAGE (BLKS) IF SUCCESS_CODE <> VALID THEN RETURN FI UPDATE_ALIAS_TABLE(ENTRY_#, SIZE, CLASS, PAGE_TABLE_LOC) SUCCESS_CODE := WRITE_ALIAS_TABLE (G_AST[PAR_INDEX].ALIAS_TABLE_LOC) IF SUCCESS_CODE <> VALID THEN RETURN ELSE SUCCESS_CODE := SEG_CREATED FI END CREATE_ENTRY

Figure 21: Create Entry Pseudo-code
When invoked, Create_Entry will determine which state the mentor segment is in (viz., if it has an alias table). If an alias table does not exist for the mentor segment, one is created and the alias table of the mentor segment's parent is updated. The alias table is read into core and a duplicate entry check is made. If no duplicate entry exists, the segment size is converted from bytes to blocks, and the secondary storage is allocated for non-zero sized segments. The appropriate data is entered into the alias table and the alias table is then written back to secondary storage.

2. **Delete an Alias Table Entry**

Delete_Entry is invoked when a user desires to delete a segment. A segment is deleted by deallocating secondary storage, and by removing the appropriate entry from the alias table of its mentor segment (the reverse logic of Create_Entry). This implies that the mentor segment must be active at the time of deletion. There are three conditions that can be encountered during the deletion of a segment: the segment to be deleted may be an inactive leaf segment, an active leaf segment, or a mentor segment.

If the segment to be deleted is an inactive leaf segment (viz., has been swapped out of core, and does not have an entry in the G_AST), the secondary storage can be deallocated and the entry deleted from the mentor segment's alias table. If the segment is an active leaf segment, the segment
must first be swapped out of core and deactivated before it can be deleted. This entails signalling the memory manager of each processor, in which the segment is active, to swap out and deactivate the segment.

If the segment to be deleted is a mentor segment, an alias table exists for that segment. If the alias table is empty, the secondary storage for the alias table and the segment can be deallocated, and the entry for the deleted segment can be removed from its mentor's alias table. If the alias table contains any entries, the segment cannot be deleted because these entries would be lost. If this condition is encountered a success code of "leaf_segment_exists" is returned to the process which requested to delete the entry. Due to a confinement problem in "upgraded" segments, this success code cannot always be passed outside of the kernel. This implies that the segment manager must strictly prohibit deletion of a segment with an access class not equal to that of the process.

The pseudo-code for DELETE_ENTRYPROCEDURE is presented in Figure 22. The parameters that are passed to this procedure are the parent's index into the G_AST and the entry number into the parent's alias table of the segment to be deleted. The alias_table_loc field is checked to determine the state of the mentor segment (either a leaf or a node), and the appropriate action is then taken. A success code is returned to indicate the results of this procedure.
DELETE_ENTRY PROCEDURE (PAR_INDEX WORD, ENTRY_# WORD)
RETURNS (SUCCESS_CODE BYTE)
LOCAL PAR_INDEX WORD ENTRY
! Check if the passed mentor segment has an alias table.!
IF G_AST[PAR_INDEX].ALIAS_TABLE_LOC <> NULL
SUCCESS_CODE := READ_ALIAS_TABLE (G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
ELSE
SUCCESS_CODE := NO_CHILD_TO_DELETE
FI
IF SUCCESS_CODE <> VALID THEN RETURN
FI
! Determine if segment has children in alias table!
IF ALIAS_TABLE_NOT_EMPTY THEN
SUCCESS_CODE := LEAF_SEGMENT_EXISTS
RETURN ! Deletion will delete children!
ELSE
! Search G_AST with UNIQUE_ID to verify segment inactive!
IF ACTIVE_IN_G_AST THEN
! Check if active in AST!
IF ACTIVE_IN_L_AST THEN
DEACTIVATE_ALL (G_AST_INDEX, L_AST_INDEX)
FI
! Check G_AST to verify segment inactive in other L_AST's!
IF ACTIVE_IN_OTHER_L_AST THEN
SIGNAL_TO_DEACTIVATE_ALL (G_AST_INDEX)
FI
FREE_SEC_STORAGE_OF_SEG & ALIAS_IF_EXISTS
DELETE_ALIAS_TABLE_ENTRY
FI
DELETE_ALIAS_TABLE_ENTRY
SUCCESS_CODE := WRITE_ALIAS_TABLE (G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
IF SUCCESS_CODE = VALID THEN
SUCCESS_CODE := SEG_DELETED
FI
END DELETE_ENTRY

Figure 22: Delete Entry Pseudo-code
3. **Activate a Segment**

Activate is invoked when a user desires to make a segment known by adding a segment to his address space. A segment is activated by making an entry into the L_AST for that processor, and the G_AST. The activated segment could be in one of three states; it could have previously been activated by another process and have a current entry in both the G_AST and L_AST, it could have previously been activated by another process on a different processor and have an entry in the G_AST but not the L_AST, or it could be inactive and have an entry in neither the G_AST nor the L_AST.

If the segment to be activated already has entries in both the L_AST and G_AST, these entries need only be updated to indicate that another process has activated the segment. The segment number is entered into the Segment_No/Access_Auth field of the L_AST, and if the segment is a leaf, its mentor's No_Active_Dependents field in the G_AST is incremented. In this design, the G_AST is always searched to determine if the segment has been previously activated by another process.

If the segment to be activated has an entry in the G_AST but not the L_AST, an entry must be made in the L_AST and the G_AST must be updated. The L_AST is searched to determine an available index. The segment number is entered into the L_AST, and the index number is entered into the G_AST.
Processes_L_ASTE_# field. If the segment to be activated is a leaf segment, its mentor's No_Active_Dependents field in the G_AST is incremented.

If the activated segment does not have an entry in either the G_AST or L_AST, an entry must be made in both. The G_AST is searched to find an available index, and the entry is made. The L_AST is then searched to find an available index, and the entry is made. The L_AST index is then entered into the G_AST Processes_L_ASTE_# field. If the activated segment is a leaf, the No_Active_Dependents field of its mentor's G_AST entry is incremented.

The pseudo-code for ACTIVATE PROCEDURE is presented in Figure 23. The parameters that are passed are the DBR_# of the signalling process, the mentor segment's index into the G_AST, the alias table entry number, and the segment number of the activated segment. The mentor segment is always checked to determine if it has an associated alias table. If it does not, the success code of "alias_does_not_exist" is returned. If the alias table does exist, it is read into core and the entry number is used as an index to obtain the activated segment's unique_id. The G_AST is then searched to determine if the segment has already been activated. If the unique_id is found, the G_AST is updated and the L_AST is either updated or an entry is made (depending on whether an entry existed or not). If the unique_id of the segment was not found during the search of the G_AST, an entry must
be made in both the G_AST and L_AST. Activate returns the activated segment's classification, size, and handle to the signalling process.
ACTIVATE PROCEDURE (DBR_# BYTE, PAR_INDEX WORD,
   ENTRY_# WORD, SEGMENT_NO BYTE)
RETURNS (SUCCESS_CODE BYTE, RET_G_AST_HANDLE HANDLE,
   CLASS BYTE, SIZE WORD)
LOCAL G_INDEX WORD, L_INDEX WORD
ENTRY
! Verify that passed segment is a mentor segment !
IF G_AST[PAR_INDEX].ALIAS_TABLE_LOC <> 0 THEN
   SUCCESS_CODE := READ_ALIAS_TABLE (G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
ELSE
   SUCCESS_CODE := ALIAS_DOES_NOT_EXIST
FI
IF SUCCESS_CODE <> VALID THEN RETURN
FI
! Check G_AST to determine if active !
SUCCESS_CODE, INDEX := SEARCH_G_AST (UNIQUE_ID)
IF SUCCESS_CODE = FOUND THEN
   IF SEGMENT_IN_L_AST THEN
      UPDATE_L_AST (SEGMENT_NO)
   ELSE
      MAKE_L_AST_ENTRY (DBR_#, SEGMENT_NO)
      UPDATE_G_AST (L_INDEX)
      IF G_AST[INDEX].ALIAS_TABLE_LOC = NULL THEN
         G_AST[PAR_INDEX].NO_DEPENDENTS_ACTIVE += 1
      FI
   FI
ELSE
   MAKE_G_AST_ENTRY (ENTRY_#)
   MAKE_L_AST_ENTRY (PAR_INDEX, ENTRY_#)
FI
SUCCESS_CODE := SEG_ACTIVATED
END ACTIVATE

Figure 23: Activate Pseudo-code
4. **Deactivate a Segment**

Deactivate is invoked when a user desires to remove a segment from his address space. To deactivate a segment, the memory manager either removes or updates an entry in both the L_AST and G_AST. Deactivate uses the reverse logic of activate. Once a segment is deactivated, it can only be reactivated via its mentor's alias table as discussed in activate. If a process requests to deactivate a segment which has not been swapped out of the process' virtual core, the memory manager swaps the segment out and updates the ABU image before the segment is deactivated. The segment to be deactivated could be in one of three states: more than one process could concurrently hold the segment active in the L_AST, the segment could be held active by one process in the L_AST and more than one in the G_AST, the segment could be held active by only one process in both the L_AST and the G_AST.

Deactivation of leaf segments and mentor segments are handled differently. If the segment is a mentor segment and has active dependents, it cannot be removed from the G_AST (even though no process currently has that segment active). This is based on the design decision which requires that the mentor of all active leaf segments remain in the G_AST to allow access to its alias table. The mentor's alias table must be accessible when an alias table is created for a de-
dependent leaf segment. If a leaf segment is deactivated, the No_Active_Dependents field of its mentor's G_AST entry is decremented. A mentor segment can only be removed from the G_AST if no process holds it active, and it has no active dependents.

If more than one process concurrently hold a segment active in the L_AST, and one of them signals to deactivate that segment, the entry in the L_AST is updated. This is accomplished by nulling out the Segment_No/Access_Auth field of the L_AST for the appropriate process. If required, the No_Active_Dependents field of its mentor segment's G_AST entry is decremented.

If only one process holds the segment active in the L_AST, and that Process signals to deactivate the segment, the L_AST entry for that segment is removed. The Processors_L_AST_E# is updated and checked to determine if there are other connected processors. If there are no other connected processors and the segment has no active dependents, the segment is removed from the G_AST. If there are other connected processors, the G_AST is updated. If the deactivated segment is a leaf, the mentor segment's No_Active_Dependents field in the G_AST is decremented.

The pseudo-code for DEACTIVATE PROCEDURE is presented in Figure 24. The parameters that are passed to the memory manager are the DBR_# of the signalling process, and the index into the G_AST for the segment to be deactivated. The
procedure first updates the L_AST, and then removes the entry if no local process holds the segment active. The G_AST is then updated, and its mentor segment is checked (if the deactivated segment was a leaf), to determine if it can be removed. If no processes currently hold the segment active, and it has no active dependents, the segment is removed from the G_AST.
DEACTIVATE PROCEDURE (DBR_#, BYTE, PAR_INDEX WORD)
RETURNS (SUCCESS_CODE BYTE)
LOCAL INDEX WORD
ENTRY
! Check if segment is in core !
IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY <> 0 THEN
! Check MNU image to determine if in local memory !
IF IN_LOCAL_MEMORY THEN
SUCCESS_CODE := OUT (DBR_#, INDEX)
FI
FI
! Remove process segment_no entry in L_AST !
L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH[DBR_] = 0
CHECK_IF_ACTIVE_IN_L_AST (L_AST_INDEX)
IF NOT_ACTIVE_IN_L_AST THEN
L_AST[L_INDEX].MEMORY_ADDR := AVAILABLE
FI
! Check if deleted segment was a leaf !
IF G_AST[INDEX].G_AST_#_PAR <> 0 THEN
G_AST[PAR_INDEX].NO_DEPENDENTS_ACTIVE -= 1
! Determine if parent can be removed !
CHECK_FOR_REMOVAL (PAR_INDEX)
FI
! Determine if deactivated segment can be removed !
CHECK_FOR_REMOVAL (INDEX)
SUCCESS_CODE := SEG_DEACTIVATED
END DEACTIVATE

Figure 24: Deactivate Pseudo-code
5. **Swap a Segment In**

SWAP_IN is invoked when a user desires to swap a segment into main memory (global or local) from secondary storage. A segment is swapped into main memory by obtaining the secondary storage location of its page table from the G_AST, allocating the required amount of main memory, and reading the segment into the allocated main memory. The segment must be active before it can be swapped into core, and the required main memory space must be available. Three conditions can be encountered during the invocation of SWAP_IN. The segment can already be located in global memory, the segment can already be located in one or more local memories, or the segment may only reside in secondary storage.

If the segment is not in local or global memory, local memory is allocated, the segment is read into the allocated memory, and the appropriate entries are made in the MMU image, the L_AST and the G_AST. If the segment is already in global memory, it can be assumed that the segment is shared and writable. In this case the only required actions are to update the G_AST and L_AST. The No_Active_In_Memory field of the G_AST entry is incremented, and the MMU image is updated to reflect the swapped in segment's core address and attributes.

If the segment already resides in one or more local memories, it must be determined if the segment is "shared" and
"writable". A segment is "shared" if it exists in more than one local memory. A segment is "writable" if one process has write access to that segment. If the segment is not shared or not writable and in local memory, the appropriate entries are updated in the MMU image, the L_AST, and the G_AST. If the segment does not reside in local memory, the required amount of local memory is allocated, the segment is read into the allocated memory, and the appropriate entries are made in the MMU image, the L_AST, and the G_AST.

If the segment is shared, writable, and in local memory, the segment must be moved to global memory. If the segment is not in the memory manager's local memory, it signals another memory manager to move the segment to global memory. After the segment is moved to global memory, the memory manager signals all of the connected memory manager's to update their L_AST and MMU data bases. When all local data bases are current, the memory manager updates the G_AST and returns a success code of seg_activated.

The pseudo-code for SWAP_IN PROCEDURE is presented in Figure 25. The arguments passed to SWAP_IN are the G_AST_INDEX of the segment to be moved in, the process' DBR#, and the access authorized. SWAP_IN will convert the segment size from bytes to blocks, and verify that the process' core will not be exceeded. If the virtual core will be exceeded, a success code of "core_space_exceeded" will be returned. If write access is permitted, the writable bit is
set. Checks are then performed to determine the segment's storage location (local or global), and the appropriate action is taken.
SWAP_IN PROCEDURE (INDEX WORD, DBR_# BYTE, ACCESS_AUTH BYTE)
RETURNS (SUCCESS_CODE BYTE)
LOCAL L_INDEX WORD, BLKS WORD
ENTRY
BLKS := CALCULATE_NO_OF_BLKS (G_AST[INDEX].SIZE)
SUCCESS_CODE := CHECK_MAX_LINEAR_CORE (BLKS)
IF SUCCESS_CODE = VIRTUAL_LINEAR_CORE_FULL THEN
  RETURN
PI
G_AST[INDEX].NO_SEGMENTS_IN_MEMORY += 1
IF ACCESS_AUTH = WRITE THEN
  G_AST[INDEX].FLAG_BITS := WRITABLE_BIT_SET
PI
! Determine if segment can be put in local memory!
IF G_AST[INDEX].FLAG_BITS AND WRITABLE_MASK = 0
  OR IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY <= 1 THEN
    ! Determine if already in local memory!
    CHECK_LOCAL_MEMORY (L_AST_INDEX)
    IF NOT_IN_LOCAL_MEMORY THEN
      ALLOCATE_LOCAL_MEMORY (BLKS)
      READ_SEGMENT (PAGE_TABLE_LOC, BASE_ADDR)
      L_AST[L_INDEX] := BASE_ADDR
    PI
    ELSE
      IF NOT_IN_GLOBAL_MEMORY THEN
        UPDATE_MNU
        UPDATE_L_AST
        RETURN
      ELSE
        ALLOCATE_GLOBAL_MEMORY (BLKS)
        IF IN_LOCAL_MEMORY THEN
          MOVE_TO_GLOBAL (L_INDEX, BASE_ADDR, SIZE)
        ELSE
          SIGNAL_OTHER_MEMORY_MANAGERS(INDEX, BASE_ADDR)
        PI
      PI
      UPDATE_MNU_IMAGE (DBR_#, SEG_#, BASE_ADDR, ACCESS, BLKS)
      UPDATE_L_AST_ACCESS (L_INDEX, ACCESS, DBR_#)
      SUCCESS_CODE := SWAPPED_IN
END SWAP_IN

Figure 25: Swap_In Pseudo-code
6. **Swap a Segment Out**

SWAP_OUT is invoked when a user desires to move a segment out of core. A segment is swapped out of core by obtaining its secondary storage location, writing the segment to that location (if required), and deallocating the main memory used. The decision to write the segment is determined by the G_AST written bit. This bit is set whenever the segment has been modified. The segment to be swapped out can be in one of two states: the segment can be in local memory, or the segment can be in global memory.

If one process has the segment in local memory and the written bit is set, the segment is written into secondary storage and the local memory is deallocated. If the written bit is not set, the local memory need only be deallocated. If more than one process has the segment in the same local memory, the segment remains in core. The appropriate MMU image is updated to reflect the segment’s deletion and the G_AST No_Active_In_Memory field is decremented.

All segments in global memory are shared and writable. If a process requests the segment to be swapped out, the segment remains in memory. The MMU image is updated to reflect the segment’s deletion, and the G_AST No_Active_In_Memory field is decremented. If the No_Active_In_Memory indicates that one process has the segment in core, its memory manager is signalled to move the segment to local memory.
The pseudo-code for SWAP_OUT PROCEDURE is presented in Figure 26. The arguments passed to SWAP_OUT are the DBR_# of the signalling process, and the G_AST_INDEX of the segment to be removed. The return parameter is a success code. SWAP_OUT removes the segment from the process’s virtual core, deletes the segment from its MMU image, and decrements the No_Active_In_Memory field. If the segment can be removed from memory, it is determined which memory can be deallocated. If the segment has been modified, it is written back to secondary storage and the appropriate memory deallocated. If the segment has not been modified, the appropriate memory is deallocated. If after the deletion one process has the segment in global memory, its memory manager need only be signalled to move the segment to local memory. When SWAP_OUT successfully completes, it returns a success code of “swapped out.”
SWAP_OUT PROCEDURE (DBR # BYTE, INDEX WORD)
RETURNS (SUCCESS_CODE BYTE)
ENTRY
BLKS := G_AST[INDEX].SIZE / BLK_SIZE
FREE_PROCESS_LINEAR_CORE (BLKS)
DELETE_MNU_ENTRY (DBR #, SEG #)
G_AST[INDEX].NO_SEGMENTS_IN_MEMORY -= 1
! Determine if segment has been written into!
IF MNU_IMAGE[DBR #].SDR(SEG #).ATTRIBUTES=WRITTEN THEN
! If segment has been written into, update G_AST!
G_AST[INDEX].FLAG_BITS := WRITTEN
FI
! Determine if segment is in global memory!
IF G_AST[INDEX].GLOBAL_ADDR <> NULL THEN
IF G_AST[INDEX].NO_SEGMENTS_IN_MEMORY = 0
AND IF G_AST[INDEX].FLAG_BITS = WRITTEN THEN
WRITE_SEG (PAGE_TABLE_LOC, MEMORY_ADDR)
FREE_LOCAL_BIT_MAP (MEMORY_ADDR, BLKS)
ELSE
IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0 THEN
FREE_LOCAL_BIT_MAP (MEMORY_ADDR, BLKS)
FI
FI
ELSE ! If not in global memory!
IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0
AND IF G_AST[INDEX].FLAG_BITS = WRITTEN THEN
WRITE_SEG (PAGE_TABLE_LOC, GLOBAL_ADDR)
FREE_GLOBAL_BIT_MAP (GLOBAL_ADDR, BLKS)
ELSE
IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0 THEN
FREE_GLOBAL_BIT_MAP (GLOBAL_ADDR, BLKS)
FI
FI
SUCCESS_CODE := SWAPPED_OUT
END SWAP_OUT

Figure 26: Swap_Out Pseudo-code
7. **Deactivate All Segments**

DEACTIVATE_ALL is invoked when it becomes necessary to remove a segment from every process' address space. Each process is checked to determine if the segment is active. If a process has the segment active, it is deactivated from its address space. The pseudo code for Deactivate_all is illustrated in Figure 27. The parameters passed to Deactivate_all are the deactivated segment's G_AST index and the L_AST index. The L_AST is searched by DBR_# to determine which process has the segment active. If the check reveals that the segment is active, it is deactivated by calling Deactivate. If the segment was successfully deactivated from all processes, a success_code of valid is returned.
DEACTIVATE_ALL 

PROCEDURE (INDEX WORD, L_INDEX WORD) 
RETURNS (SUCCESS_CODE BYTE) 
ENTRY 
LOCAL I BYTE 
I := 0 
DO 
IF I = MAX_DBR # THEN 
EXIT 
FI 
IF LAST[ L_INDEX].SEGMENT_NO/ACCESS_AUTH[I] <> ZERO THEN 
SUCCESS_CODE := DEACTIVATE (I, INDEX) 
IF SUCCESS_CODE <> SEG_DEACTIVATED THEN 
RETURN 
FI 
FI 
I += 1 
OD 
SUCCESS_CODE := VALID 
END DEACTIVATE_ALL 

Figure 27: Deactivate All Pseudo-code 

8. Move a Segment to Global Memory 

MOVE_TO_GLOBAL is invoked when it becomes necessary to move a segment from local to global memory. If a segment resides in one or more local memories, and a process with write access swaps that segment into core, or if a segment resides in in local memory (with write access) and another process with read access requests the segment swapped in, the segment is moved from a local to global memory to avoid a secondary storage access. If the segment resides in the running memory manager's local memory, it will affect the
segment transfer, otherwise it will signal another memory manager of a connected processor to affect the transfer. Figure 28 illustrates the pseudo-code for MOVE_TO_GLOBAL.

Once the segment has been moved to global memory, the signalled memory manager will update the MMU images for all connected processes, and deallocate the freed local memory. A success code of completed will be returned to the signalling memory manager. The parameters passed to the memory manager are the segment's L_AST index, the global memory address of the move, and the size of the segment. This information is passed because the G_AST is locked during this request.

MOVE_TO_GLOBAL PROCEDURE (L_INDEX WORD, GLOBAL_ADDR WORD, SIZE WORD)
    RETURNS (SUCCESS_CODE BYTE)
ENTRY
    ! Move segment from local memory to global memory!
    DO_MEMORY_MOVE (MEMORY_ADDR, GLOBAL_ADDR)
    L_AST[L_INDEX].MEMORY_ADDR := AVAILABLE
    ! Update the MMU image to reflect new address!
    DO FOR_ALL DBR'S
        IF L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH <> 0 ANDIF
        MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=IN_LOCAL THEN
            MMU_IMAGE[DBR_#].SDR[SEG_#].BASE_ADDR:=GLOBAL_ADDR
        FI
    OD
    SUCCESS_CODE := VALID
END MOVE_TO_GLOBAL

Figure 28: Move To Global Pseudo-code
9. **Move a Segment to Local Memory**

MOVE_TO_LOCAL is invoked when it becomes necessary to move a segment from global to local memory. This occurs when one of two processes which hold a segment in global memory swaps the segment out. The segment is moved from global memory to the local memory of the remaining process. Figure 29 illustrates the pseudo-code for MOVE_TO_LOCAL. The parameters passed to the memory manager are the segment's LAST index, the global address of the segment, and the size of the segment. The return parameter is a success code. The MNV images of the signalled process are updated after the move has been made, and the global memory is deallocated.

MOVE_TO_LOCAL PROCEDURE (L_INDEX WORD, GLOBAL_ADDR WORD, SIZE WORD)
RETURNS (SUCCESS_CODE BYTE)
ENTRY
BLKS := SIZE / BLK_SIZE
BASE_ADDRESS := ALLOCATE_LOCAL_MEMORY (BLKS)
! Move from global to local memory!
MEMORY_MOVE (GLOBAL_ADDR, BASE_ADDRESS, SIZE)
L_LAST[L_INDEX].MEMORY_ADDR := BASE_ADDRESS
DO FOR ALL DBR'S
IF LAST[L_INDEX].SEGMENT_NO/ACCESS_AUTH <> 0 AND IF
MNU_IMAGE[DBR #].SDR[SEG #].ATTRIBUTES=IN_LOCAL THEN
MNU_IMAGE[DBR #].SDR[SEG #].BASE_ADDR:=BASE_ADDRESS
FI
OD
SUCCESS_CODE := VALID
END MOVE_TO_LOCAL

Figure 29: Move To Local Pseudo-code
10. **Update the MMU Image**

UPDATE is invoked following a MOVE_TO_GLOBAL operation. After a segment has been moved from local memory to global memory, it is necessary to signal the memory managers of all connected processors to update their MMU images and L_AST with the current location of the segment. They must also deallocate the moved segment's local memory. Figure 30 illustrates the pseudo-code of UPDATE. The parameters passed to the memory manager are the segment's L_AST index, the new global address for the segment, and the size of the segment. The return parameter is a success code.

```
UPDATE PROCEDURE (L_INDEX WORD, GLOBAL_ADDR WORD, SIZE WORD)
RETURNS (SUCCESS_CODE BYTE)
ENTRY
DO FOR_ALL_DBR'S
   IF L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH <> 0 AND IF
      MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=IN_LOCAL THEN
         MMU_IMAGE[DBR_#].SDR[SEG_#].BASE_ADDR :=
         GLOBAL_ADDR
   FI
OD
BLKS := SIZE / BLK_SIZE
FREE_LOCAL_BIT_MAP (MEMORY_ADDR, BLKS)
L_AST[L_INDEX].MEMORY_ADDR := ACTIVE
SUCCESS_CODE := VALID
END UPDATE
```

Figure 30: Update Pseudo-code
E. SUMMARY

In this chapter the detailed design of the memory manager process has been presented. The purpose of the memory manager was outlined, followed by a detailed discussion of the memory manager's data bases. The design presented has identified ten basic functions for the memory manager. The success codes returned by the memory manager are presented in Figure 31.

This design has assumed that the kernel level inter-process synchronization primitives will be Saltzer's signal and wait primitives [14]. This fact dominated the design decision to lock the G_AST in the user's process before it signals the memory manager. In a multi-processor environment, the possibility of a deadly embrace exists if the memory manager processes lock the G_AST. Should follow on work implement eventcounts and sequencers as kernel level synchronization primitives, the locking of the G_AST and memory manager synchronization will need to be readdressed.
SYSTEM WIDE

INVALID
SWAPPED_IN
SWAPPED_OUT
SEG_ACTIVATED
SEG_DEACTIVATED
SEG_CREATED
SEG_DELETED
VIRTUAL_CORE_FULL
DUPLICATE_ENTRY
READ_ERROR
WRITE_ERROR
DRIVE_NOT_READY

KERNEL LOCAL

LEAF_SEGMENT_EXISTS
NO_LEAF_EXISTS
ALIAS_DOES_NOT_EXIST
NO_CHILD_TO_DELETE
G_LAST_FULL
L_LAST_FULL
LOCAL_MEMORY_FULL
GLOBAL_MEMORY_FULL
SECONDARY_STORAGE_FULL

MEMORY MANAGER LOCAL

VALID
INVALID
FOUND
NOT_FOUND
IN_LOCAL_MEMORY
NOT_IN_LOCAL_MEMORY
! + DISK ERRORS !

Figure 31: Success Codes
A. CONCLUSIONS

The memory manager design utilized state of the art software techniques and hardware devices. The design was developed based upon ZILOG'S Z8001 sixteen bit segmented microprocessor used in conjunction with the Z8010 Memory Management Unit [23]. A microprocessor which supports segmentation is required to provide access control of the stored data. The actual implementation of the selected thread was conducted upon the Z8002 non-segmented microprocessor without the Z8010 MMU.

While information security requires that the microprocessor support segmentation, the memory manager was developed to be configuration independent. The design will support a multi-processor environment, and can be easily implemented upon any microprocessor or secondary storage device. The loop free modular design facilitates any required expansion or modification.

Global bus contention is minimized by the memory manager. Segments are stored in global memory only if they are shared and writable. Secondary storage is accessed only if
the segment does not currently reside in global memory or some local memory. The controlled sharing of segments optimizes main memory usage.

The storage of the alias tables in secondary storage supports the recreation of user file hierarchies following a system crash. The aliasing scheme used to address segments supports system security by not allowing the segment’s memory location or unique identification to leave the memory manager.

The design of the distributed kernel was clarified by assigning the MMU image management to the memory manager. The transfer of responsibility for memory allocation and deallocation from the supervisor to the memory manager provides support for dynamic memory management.

In conclusion, the memory manager process will securely manage segments in a multi-processor environment. The process is efficient, and is configuration independent. The primitives provided by the memory manager will support the construction of any desired supervisor/user process built upon the kernel.
B. FOLLOW ON WORK

There are several possible areas in the SASS design that can be looked into for continued research. The complete implementation of the memory manager design (refine and optimize the current PLZ/SYS code) is one possibility. Other possibilities include the implementation of dynamic memory management, and modifying the interface of the memory manager with the distributed kernel using eventcounts and sequencers for inter-process communication.

The implementation of the supervisor has not been addressed to date. Areas of research include the implementation of the file manager and input/output processes, and the complete design and implementation of the user-host protocols. The implementation of the gatekeeper, and system initialization are other possible research areas. Dynamic process creation and deletion, and the introduction of multi-level hosts could also prove interesting.
PART D

AN IMPLEMENTATION OF MULTIPROGRAMMING AND PROCESS MANAGEMENT FOR A SECURITY KERNEL OPERATING SYSTEM

This section contains updated excerpts from a Naval Postgraduate School MS Thesis by S. L. Reitz [12]. The origins of these excerpts are:

- **INTRODUCTION** from Chapter I
- **IMPLEMENTATION** from Chapter IV
- **CONCLUSION** from Chapter V

Minor changes have been made for integration into this report.
Chapter XIII

INTRODUCTION

The application of contemporary microprocessor technology to the design of large-scale multiple processor systems offers many potential benefits. The cost of high-power computer systems could be reduced drastically; fault tolerance in critical real-time systems could be improved; and computer services could be applied in areas where their use is not now cost effective. Designing such systems presents many formidable problems that have not been solved by the specialized single processor systems available today.

Specifically, there is an increasing demand for computer systems that provide protected storage and controlled access for sensitive information to be shared among a wide range of users. Data controlled by the Privacy Act, classified Department of Defence (DoD) information, and the transactions of financial institutions are but a few of the areas which require protection for multiple levels of sensitive information. Multiple processor systems which share data are well suited to providing such services - if the data security problem can be solved.

A solution to these problems - a multiprocessor system design with verifiable information security - is offered in
a family of secure, distributed multi-microprocessor operating systems designed by O'Connell and Richardson [7]. A subset of this family, the Secure Archival Storage System (SASS) [9,5], has been selected as a testbed for the general design. SASS will provide consolidated file storage for a network of possibly dissimilar "host" computers. The system will provide controlled, shared access to multiple levels of sensitive information (Figure 32).

This thesis presents an implementation of a basic monitor for the O'Connell-Richardson family of operating systems. The monitor provides multiprogramming and process management functions specifically addressed to the control of physical processor resources of SASS. Concurrent thesis work [7] is developing a detailed design for a security kernel process, the Memory Manager, which will manage SASS memory resources.
Figure 32: SASS SYSTEM
Chapter XIV
IMPLEMENTATION

Implementation of the distributed kernel was simplified by the hierarchical structure of the design for it permitted methodical bottom-up construction of a series of extended machines. This approach was particularly useful in this implementation since the bare machine, the Z8000 Developmental Module, was provided with only a small amount of software support.

A. DEVELOPMENTAL SUPPORT

A Zilog MCZ Developmental System provided support in developing Z8000 machine code. It provided floppy disk file management, a text editor, a linker and a loader that created an image of each Z8000 load module.

A Z8000 Developmental Module (DM) provided the necessary hardware support for operation of a Z8002 non-segmented microprocessor and 16K words (32K bytes) of dynamic RAM. It included a clock, a USART, serial and parallel I/O support, and a 2K PROM monitor.

The monitor provided access to processor registers and memory, single step and break point functions, basic I/O functions, and a download/upload capability with the MCZ system.
Since a segmented version of the processor was not available for system development, segmentation hardware was simulated in software as an MNU image (see Figure 33). Although this data structure did not provide the hardware support (traps) required to protect segments of the kernel domain, it preserved the general structure of the design.

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>High byte</td>
<td>Low byte</td>
</tr>
<tr>
<td>seg #</td>
<td>V</td>
</tr>
</tbody>
</table>

Figure 33: MNU_IMAGE

- 113 -
B. INNER TRAFFIC CONTROLLER

The Inner Traffic Controller runs on the bare machine to create a virtual environment for the remainder of the system. Only this module is dependent on the physical processor configuration of the system. All higher levels see only a set of running virtual processors. A kernel data base, the Virtual Processor Table is used by the Inner Traffic Controller to create the virtual environment of this first level extended machine. A source listing of the Inner Traffic Controller module is contained in Appendix G.

1. Virtual Processor Table (VPT)

The VPT is a data structure of arrays and records that maintains the data used by the Inner Traffic Controller to multiplex virtual processors on a real processor and to create the extended instruction set that controls virtual processor operation (see Figure 34). There is one table for each physical processor in the system. Since this implementation was for a uniprocessor system (the Z8000 DM), only one table was necessary.

The table contains a LOCK which supports an exclusion mechanism for a multiprocessor system. It was provided in this implementation only to preserve the generality of the design.

The Descriptor Base Register (DBR) binds a process to a virtual processor. The DBR points to an MMU_IMAGE contain-
A virtual processor (VP) can be in one of three states: running, ready, and waiting (Figure 35). A running VP is currently scheduled on a real processor. A ready VP is ready to be scheduled when selected by the level-1 schedule.
ing algorithm. A waiting VP is awaiting a message from some other VP to place it in the ready list. In the meantime it is not in contention for the real processor.
Figure 35: Virtual Processor States
2. **Level-1 Scheduling**

Virtual processor state changes are initiated by the inter-virtual-processor communication mechanisms, SIGNAL and WAIT. These level-1 instructions implement the scheduling policy by determining what virtual processor to bind to the real processor. The actual binding and unbinding is performed by a Processor switching mechanism called SWAP_DBR [14]. Processor switching implies that somehow the execution point and address space of a new process are acquired by the processor. Care must be taken to ensure that the old process is saved and the new process loaded in an orderly manner. A solution to this problem, suggested by Saltzer [14], is to design the switching mechanism so that it is a common procedure having the same segment number in every address space.

In this implementation a processor register (R14) was reserved within the switching mechanism for use as a DBR. Processor switching was performed by saving the old execution point (i.e., processor registers and flag control word), loading the new DBR and then loading the new execution point. The processor switch occurs at the instant the DBR is changed (see Figure 36). Because the switching procedure is distributed in the same numbered segment in all address spaces, the "next" instruction at the instant of the switch will have the same offset no matter what address
space the processor is in. This is the key to the proper operation of SWAP_DBR.

To convert this switching mechanism to segmented hardware it is necessary merely to replace SWAP_DBR with special I/O block-move instructions that save the contents of the MMU in the appropriate MMU_IMAGE and load the contents of the new MMU_IMAGE into the MMU.

a. Getwork

SWAP_DBR is contained within an internal Inner Traffic Controller procedure called GETWORK. In addition to multiplexing virtual processors on the CPU, GETWORK interprets the virtual processor status flags, IDLE and PREEMPT, and modifies VP scheduling accordingly in an attempt to keep the CPU busy doing useful work.

There are actually two classes of idle processes within the system. One class belongs to the Traffic Controller. Conceptually there is a ready level-2 idle process for each virtual processor available to the Traffic Controller for scheduling. When a running process blocks itself, the Traffic Controller schedules the first ready process. This will be an idle process if no supervisor processes are in the ready list.

The second class of idle process exists in the kernel. The kernel Idle process is permanently bound to the lowest priority virtual processor.
<table>
<thead>
<tr>
<th>Process #1 Address space</th>
<th>Process #2 Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Call SWAP_DBR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Save return point  on call stack. (Process #1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Save execution point</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap DBR (R14) ----------------- &gt; Swap DBR (R14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>processor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>switch</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load new execution point.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Load return point from call stack (process #2)</td>
</tr>
</tbody>
</table>

**Figure 36: SWAP_DBR**
The distinction is made between these classes because of the need to keep the CPU busy doing useful work whenever possible. There is no need for GETWORK to schedule a level-2 idle process that has been loaded on a virtual processor, because the idle process does no useful work. The virtual processor IDLE_FLAG indicates that a virtual processor has been loaded with a level-2 idle process. GETWORK will schedule this virtual processor only if the PREEMPT flag is also set. The PREEMPT flag is a signal from the Traffic Controller that a supervisor process is now ready to run.

When GETWORK can find no other ready virtual processors with IDLE and PREEMPT flags off, it will select the virtual processor permanently bound to the kernel Idle process. Only then will the Idle process actually run on the CPU.

Getwork contains two entry points. The first, a normal entry, resets the preempt interrupt return flag. (R0 is reserved for this purpose within GETWORK.) The second, a hardware interrupt entry point, contains an interrupt handler which sets the preempt interrupt return flag. The DSR (R14) must also be set to the current value by any procedure that calls GETWORK in order to permit the SWAP_DBR portion of GETWORK to have access to the scheduled process's address space. Upon completion of the processor switch, GETWORK examines the interrupt return flag to determine whether a normal return or an interrupt return is required.
The hardware interrupt entry point in GETWORK supports the technique used to initialize the system. Each process address space contains a kernel domain stack segment used by SWAP-DBR in GETWORK to save and restore VP states. For the same reason that SWAP-DBR is contained in a system wide segment number, the stack segment in each process address space will also have the same number (Segment #1 in this implementation). Each stack segment is initially created as though it's process had been previously preempted by a hardware interrupt. This greatly simplifies the initialization of processes at system generation time. The details of system initialization will be described later in this chapter. It is important to note here, however, that GETWORK must be able to determine whether it was invoked by a hardware preempt interrupt or by a normal call, before it can execute a return to the calling procedure. This is because a hardware interrupt causes three items to be placed on the system stack: the return location of the caller, the flag control word, and the interrupt identifier, whereas a normal call places only the return location on the stack. Therefore, in order to clean up the stack, GETWORK must execute an interrupt return (assembly instruction:IRET) if entry was via the hardware preempt handler (i.e., R0 set). This instruction will pop the three items off the stack and return to the appropriate location. If the interrupt return flag, R0, is off, a normal return is executed.
During normal operation, \texttt{SWAP-DBR} manipulates process stacks to save the old VP state and load the new VP state. This action proceeds as follows (Figure 37):

1. The Flag Control Word (FCW), the Stack Pointer (R15) and the preempt return flag (RO) are saved in the old VP's kernel stack.
2. The DBR (R14) is loaded with the new VP's DBR. This permits access to the address space of the new process.
3. The Flag Control Word (FCW), the Stack Pointer (R15) and the Interrupt Return Flag (RO), are loaded into the appropriate CPU registers.
4. RO is tested. If it is set, \texttt{GETWORK} will execute an interrupt return. If it is off, a normal return occurs.

By constructing \texttt{GETWORK} in this way, both system initialization and normal operations can be handled in the same way. A high level \texttt{GETWORK} algorithm is given in Figure 38.
Figure 37: Kernel Stack Segments
GETWORK Procedure (DBR = R14)

Begin

Reset Interrupt Return Flag (RO)

Skip hardware preempt handler

Hardware Preempt Entry:
  Set DBR
  Save CPU registers
  Save supervisor stack pointer
  Set Interrupt Return Flag (RO)

Get first ready VP

Do while not Select
  If Idle flag is set then
    if Preempt flag is set then
      select
    else
      get next ready VP
    end if
  else
    select
  end if
end do

SWAP_DBR:
  Save old VP registers in stack segment
  Swap dbr (R14)
  Load new VP registers in stack segment

  If Interrupt Return Flag is set then
    unlock VPT
  simulate GATEKEEPER exit:
    Call TEST_VPREEMPT
    Restore supervisor registers
    Restore supervisor stack pointer

  Execute Interrupt Return (IRET)
end if

Execute normal return

end GETWORK

Figure 38: GETWORK
3. **Virtual Processor Instruction Set**

The heart of the SASS scheduling mechanism is the internal procedure, `GETWORK`. It provides a powerful internal primitive for use by the virtual processors and greatly simplifies the design of the virtual processor instruction set. Virtual processor instructions perform three types of functions: multiprogramming, process management and virtual interrupts.

`SIGNAL` and `WAIT` provide synchronization and communication between virtual processors. They multiplex virtual processors on a CPU to provide multiprogramming. This implementation used a version of the signal and wait algorithms proposed by Saltzer [14]. In the SASS design each CPU is provided with a unique (fixed) set of virtual processors. The interaction among virtual processors is a result of multiprogramming them on the real processor. Only one virtual processor is able to access the VPT at a time because of the use of the VPT LOCK (SPIN_LOCK) to provide mutual exclusion. Therefore race and deadlock conditions will not develop and the signal pending switch used by Saltzer is not necessary.

This implementation also included message passing mechanism not provided by Saltzer. The message slots available for use by virtual processors are initially contained in a queue pointed to by `FREE-LIST`. When a message is sent from one VP to another, a message slot is removed from the free
list and placed in a FIFO message queue belonging to the VP receiving the message. The head of each VP's message queue is pointed to by MSG-LIST. Each message slot contains a message, the ID of the sender, and a pointer to the next message in the list (either the free list or the VP message list.

IDLE and SWAP_VDBR provide the Traffic Controller with a means of scheduling processes on the running VP.

SET_VP_PREEMPT and TEST_VP_PREEMPT install a virtual interrupt mechanism in each virtual processor. When the Traffic Controller determines that a virtual processor should give up its process because a higher priority process is now ready, it sets the PREEMPT flag in that VP. Then, even if an idle process is loaded on the VP, it will be scheduled and will be loaded with the first ready process. Test_VP_preempt is a virtual interrupt unmasking mechanism which forces a process to examine the preempt flag each time it exists from the kernel.

a. Wait

WAIT provides a means for a virtual processor to move itself from the running state to the waiting state when it has no more work to do. It is invoked only for system events that are always of short duration. It is supported by three internal Procedures.
SPIN_LOCK enables the running VP to gain control of the Virtual Processor Table. This procedure is only necessary in a multiprocessor environment. The running VP will have to wait only a short amount of time to gain control of the VPT. SPIN_LOCK returns when the VP has locked the VPT.

GETWORK loads the first eligible virtual processor of the ready list on the real processor. Before this procedure is invoked, the running VP is placed in the ready state. Both ready and running VP's are members of a FIFO queue. GETWORK selects the first VP in this ready list, loads it on the CPU, and places it in the running state. When GETWORK returns, the first VP of the queue will always be running and the second will be the first VP in the ready queue.

GET_FIRST_MESSAGE returns the first message of the message list (also managed as a FIFO queue) associated with the running VP. The action taken by WAIT is as follows:

WAIT Procedure (Returns: Msg, Sender_ID)

Begin
Lock VPT (call SPIN_LOCK)
If message list empty (i.e., no work) Then
  Move VP from Running to Waiting state
  Schedule first eligible Ready VP (call GETWORK)
end if
(NOTE: process suspended here until
- 128 -
it receives a signal and is selected by GETWORK.)

Get first message from message list (call GET-FIRST-MSG)

Unlock VPT

Return

end WAIT

If the running virtual processor calls WAIT and there is a message in its message list (placed there when another VP signaled it) it will get the message and continue to run. If the message list is empty it will place itself in the wait state, schedule the first ready virtual processor, and move it to the running state. The virtual processor will remain in the waiting state until another running VP sends it a message (via SIGNAL). It will then move to the ready list. Finally it will be selected by GETWORK, the next instructions of WAIT will be executed, it will receive the message for which it was waiting, and it will return to the caller.
b. Signal

Messages are passed between virtual processors by the instruction, SIGNAL, which uses four internal procedures, SPIN_LOCK, ENTER_MSG_LIST, MAKE_READY, and GETWORK.

SPIN_LOCK, as explained above insure that only one virtual processor has control of the Virtual Processor Table at a time.

ENTER_MSG_LIST manages a FIFO message queue for each virtual Processor and for free messages. This queue is of fixed maximum length because of the implementation decision to restrict the use of SIGNAL. A running VP can send no more than one message (SIGNAL) before it receives a reply (i.e., WAIT's for a message). Therefore if there are N virtual processors per real processors, the message queue length, L, is:

\[ L = N - 1 \]

MAKE_READY manages the virtual processor ready queue. If a message is sent to a VP in the waiting state, MAKE_READY wakes it up (it places it in the ready state) and enters it in the ready list. If a running VP signals a waiting VP of higher priority, it will place itself back in the ready state and the higher priority VP will be selected. The action taken by signal is as follows:

SIGNAL Procedure (Message, Destination_VP)

Begin
Lock VPT (call SPIN_LOCK)
Send message (call ENTER_MSG_LIST)
If signaled VP is waiting Then
   Wake it up and make it ready
      (call MAKE_READY)
end if
Put running VP in ready state.
Schedule first eligible ready VP
   (call GETWORK)
Unlock VPT
Return (Success_code)
End SIGNAL

c. SWAP_VDBR

SWAP_VDBR contains the same processor switching mechanism used in SWAP_DBR, but applies it to a virtual processor rather than a real processor. Switching is quite simple in this virtual environment because both processor execution point and address space are defined by the Descriptor Base
Register. **SWAP_VDBR** is invoked by the Traffic Controller to load a new process on a virtual processor in support of level-2 scheduling. It uses GETWORK to control the associated level-1 scheduling. The action taken by **SWAP_VDBR** is:

**SWAP_VDBR Procedure (New_DBR)**

Begin

Lock VPT (call SPIN_LOCK)

Load running VP with New_DBR

Place running VP in ready state

Schedule first eligible ready VP

(call GETWORK)

Unlock VPT

Return

End SWAP_VDBR

In this implementation one restriction is placed upon the use of this instruction. If a virtual processor's message list contains at least one message, it cannot give up its current DBR. This problem is avoided as the natural result of using SIGNAL and WAIT only for system events, and
"masking" preempts within the kernel. If this were permitted, the messages would lose their context. (The messages in a VP_MSG_LIST are actually intended for the process loaded on the VP.)

d. IDLE

The IDLE instruction loads the Idle DBR on the running virtual processor. Only virtual processors in contention for process scheduling will be loaded by this instruction. (The Traffic Controller is not even aware of virtual processors permanently bound to kernel processes.)

IDLE has the same scheduling effect as SWAP_VDBR, but it also sets the IDLE_FLAG on the scheduled VP. The distinction is made between the two cases because, although the Traffic Controller must schedule an Idle process on the VP if there are no other ready processes, the Inner Traffic Controller does not wish to schedule an Idle VP if there is an alternative. This would be a waste of physical processor resources. The setting of the IDLE_FLAG by the Traffic Controller aids the Inner Traffic Controller in making this scheduling decision. Logically, there is an idle process for each VP; actually the same address space (DBR) is used for all idle processes for the same CPU, since only one will run at a time. As previously explained, virtual processors loaded by this instruction will be selected by GETWORK only to give the Idle process away for a new process in response to a virtual preempt interrupt. The action of IDLE is:
IDLE Procedure

Begin
Lock VPT (call SPIN_LOCK)
Load running VP with Idle DBR
Set VP’s IDLE_FLAG
Place running VP in ready state
Schedule first eligible ready VP
(call GETWORK)
Unlock VPT
Return
End IDLE

e. SET_VPREEVENT

SET_VPREEVENT sets the preempt interrupt flag on a specified virtual processor. This forces the virtual processor into level-1 scheduling contention, even if it is loaded with an Idle process. The instruction retrieves an idle
virtual processor in the same way a hardware preempt retrieves an idle CPU by forcing the VP to be selected by GETWORK. The only difference between the two cases is the entry point used in GETWORK. The action of SET_VPREEMPT is:

```
SET_VPREEMPT Procedure (VP)
Begin
  Set VP's PREEMPT flag
  If VP belongs to another CPU Then
    send hardware interrupt
  end if
Return
End SET_VPREEMPT
```

Since the action is a safe sequence, no deadlocks or race conditions will arise and no lock is required on the VPT.

f. TEST_VPREEMPT
Within the kernel of a multiprocessor system all process interrupts (which excludes system I/O interrupts) are masked. If process interaction results in a virtual preempt being sent to the running virtual processor by another CPU, it will not be handled since GETWORK has already been invoked. TEST_VPREEMPT provides a virtual preempt interrupt unmasking mechanism.

TEST_VPREEMPT mimics the action of a physical CPU when interrupts are unmasked. It forces the process execution point back down into the kernel each time the process attempts to leave the kernel domain, where the preempt flag of the running VP is examined. If the flag is off, TEST_VPREEMPT returns and the execution point exits through the Gatekeeper into the supervisor domain of the process address space as described above. However, if the PREEMPT flag is on, the TEST_VPREEMPT executes a virtual interrupt handler located in the Traffic Controller. This jump from the Inner Traffic Controller to the Traffic Controller (TC_PREEMPT_HANDLER) is a close parallel to the action of a CPU in response to a hardware interrupt, that is a jump to an interrupt handler. The Traffic Controller Preempt Handler forces level-2 and level-1 scheduling to proceed in the normal manner. The preempt handler forces the Traffic Controller to examine the APT and to apply the level-2 scheduling algorithm, TC_GETWORK. If the APT has been changed since the last invocation of this scheduler, it will be re-
flected in the scheduling selections. Eventually, when the running VP's preempt flag is tested and found to be reset, TEST_VPREEMPT will return to the Gatekeeper where the process execution point will finally make a normal exit into its supervisor domain. TEST_VPREEMPT performs the following action:

TEST_VPREEMPT Procedure

Begin

Do while running VP's PREEMPT flag is set

Reset PREEMPT flag

Call preempt handler

(Call TC_PREEMPT_HANDLER)

End do

Return

End TEST_VPREEMPT
C. TRAFFIC CONTROLLER

The Traffic Controller runs in a virtual environment created by the Inner Traffic Controller. It sees a set of running virtual processor instructions: SWAP_VDBR, IDLE, SET_VPREADY, and RUNNING_VP, and provides a scheduler, TC_GETWORK, which multiplexes processes on virtual processors in response to process interaction. It also creates a level-2 instruction set: ADVANCE, AWAIT, and PROCESS_CLASS, which is available for use by higher levels of the design. The Traffic Controller uses a global data base, the ACTIVE PROCESS TABLE to support its operation.

1. Active Process Table (APT)

The Active Process Table is a system-wide kernel data base containing entries for each supervisor process in SASS (Figure 39). It is indexed by active process ID. The structure of the APT closely parallels that of the Virtual Processor Table. It contains a LOCK to support the implementation of a mutual exclusion mechanism, a RUNNING_LIST, and a READY_LIST_HEAD. The Traffic Controller is only concerned with virtual processors that can be loaded with supervisor processes. Since two VP's are permanently bound to kernel processes (the Memory Manager and the Idle Process), they cannot be in contention for level-2 scheduling; the Traffic Controller is unaware of their existence; since there are a number of available virtual processors, the
RUNNING_LIST was implemented as an array indexed by VP_ID. The READY_LIST_HEAD points to a FIFO queue that includes both running and ready processes. The running processes will be at the top of the ready list.

Because of their completely static nature, idle processes require no entries in the APT. Logically, there is an idle process at the end of the ready list for each VP available to the Traffic Controller. If the ready list is empty, TC_GETWORK loads one of these “virtual” idle processes by calling IDLE, and enters a reserved identifier, #IDL2, in the appropriate RUNNING_LIST entry. This identifier is the only data concerning idle processes that is contained in the APT. Idle process scheduling considerations are moved down to level-1, because the Inner Traffic Controller knows about physical processors, and can optimize CPU use by scheduling idle processes only when there is nothing else to do.

The subject access class, S_CLASS, provides each process with a label that is required by level-3 modules to enforce, the SASS non-discretionary security policy.
<table>
<thead>
<tr>
<th>LOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNNING_LIST PROCESS_ID</td>
</tr>
<tr>
<td>VP_ID</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>READY_LIST_HEAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBR</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>AP</td>
</tr>
<tr>
<td>Index</td>
</tr>
<tr>
<td>V</td>
</tr>
</tbody>
</table>

**Figure 39: Active Process Table**
2. **Level-2 Scheduling**

Above the Traffic Controller, SASS appears as a collection of processes in one of the three states: running, ready, or blocked. Running and ready states are analogous to the corresponding virtual processor states of the Inner Traffic Controller. However, because of the use of event-count synchronization mechanisms by the Traffic Controller, the blocked state has a slightly different connotation than the VP waiting state.

Blocked processes are waiting for the occurrence of a non-system event, e.g., the event occurrence may be signalled from the supervisor domain. When a specific event happens, all of the blocked processes that were awaiting that event are awakened and placed in the ready state. This broadcast feature of event occurrence is more powerful than the message passing mechanism of SIGNAL, which must be directed at a single recipient.

Just as SIGNAL and WAIT provide virtual processor multiplexing in level-1, the eventcount functions, ADVANCE and AWAIT, control process scheduling in level-2.

a. **TC_GETWORK**

Level-2 scheduling is implemented in the internal Traffic Controller procedure, TC_GETWORK. This procedure is invoked by eventcount functions when a process state change
may have occurred. It loads the first ready process on the currently scheduled VP (i.e., the virtual processor that has been scheduled at level-1 and is currently executing on the CPU).

TC_GETWORK Procedure
Begin
VP_ID := RUNNING_VP
Do while not end of ready list
if process is running then
get next ready process
else
RUNNING_LIST [VP_ID] := PROCESS_ID
Process state := running
SWAP_VDBR
end if
end do
If end of running list (no ready processes) Then
RUNNING_LIST := #IDLE
IDLE
end if
Return
End TC_GETWORK

- 142 -
b. **TC_PREEMPT_HANDLER**

Preempt interrupts are masked while a process is executing in the kernel domain. As the process leaves the kernel, the gatekeeper unmasks this virtual interrupt by invoking TEST_VPREEMPT. This instruction tests the scheduled VP's PREEMPT flag. If this flag is off, the process returns to the Gatekeeper and exits from the kernel; but if the flag is set, TEST_VPREEMPT calls the Traffic Controller's virtual preempt interrupt handler, TC_PREEMPT_HANDLER. This handler invokes TC_GETWORK, which re-evaluates level-2 scheduling. Eventually, when the schedulers have completed their functions, the handler will return control to the preempted process, which will return to the Gatekeeper for a normal exit. This sequence of events closely parallels the action of a hardware interrupt, but in the environment of a virtual processor rather than a CPU. The virtualization of interrupts provides the ability for one virtual processor to interrupt execution of another that may, or may not, be running on a CPU at that time. This is provided without disrupting the logical structure of the system. This capability is particularly useful in a multiprocessor environment where the target virtual processor may be executing on another CPU. Because these interrupts will be virtualized, the operating system will retain control of the system. The action of the TC_PREEMPT_HANDLER is described in the procedure below.
TCPREEMPTHANDLER Procedure

Begin

Call WAIT_LOCK

VP_ID := RUNNING_VP

Process_ID := RUNNING LIST [VP_ID]

If process is not idle Then

Process state := ready

end if

Call TC_GETWORK

Call WAIT_UNLOCK

RETURN

End TC_PREEMPT_HANDLER

WAIT_LOCK and WAIT_UNLOCK provide an exclusion mechanism which prevents simultaneous multiple use of the APT in a multiprocessor configuration. This mechanism invokes WAIT and SIGNAL of the Inner Traffic Controller.
3. **Eventcounts**

An eventcount is a non-decreasing integer associated with a global object called an event [11]. The Event Manager, a level-3 module, controls access to event data when required and provides the Traffic Controller with a HANDLE, an INSTANCE, and a COUNT. The values for all eventcounts (and sequencers) are maintained at the Memory Manager level and are accessed by calls to the Memory Manager. The HANDLE provides the traffic controller with an event ID, associated with a particular segment. INSTANCE is a more specific definition of the event. For example, each SASS supervisor segment has two eventcounts associated with it, a INSTANCE_1 and a INSTANCE_2, that the supervisor uses to keep track of read and write access to the segment [9]. Eventcounts provide information concerning system-wide events. They are manipulated by the Traffic Controller functions ADVANCE and AWAIT and by the Memory Manager functions, READ and TICKL.

A proposed high level design for ADVANCE and AWAIT is provided by Reitz [12].

**a. Advance**

ADVANCE signals the occurrence of an event (e.g., a read access to a particular supervisor segment). The value of the eventcount is the number of ADVANCE operations that have been performed on it. When an event is advanced, the fact must be broadcast to all blocked processes awaiting it and
the process must be awakened and placed on the ready list. Some of the newly awakened processes may have a higher priority than some of the running processes. In this case a virtual preempt, SET_VPREAMPT (VP_ID), must be sent to the virtual processors loaded with these lower priority processes.

b. Await

When a process desired to block itself until a particular event occurs, it invokes AWAIT. This procedure returns to the calling process when a specified eventcount is reached. Its function is similar to WAIT.

c. Read

READ returns the current value of the eventcount. This is an Event Manager (level three) function. This module calls the Memory Manager module to obtain the eventcount value.

d. Ticket

TICKET provides a complete time-ordering of possibly concurrent events. It uses a non-decreasing integer, called a sequencer, which is also associated with each supervisor segment. As with READ, this is an Event Manager function that calls the Memory Manager to access the sequencer value. Each invocation of TICKET increments the value of the se-
quencer and returns it to the caller. Two different uses of ticket will return two different values, corresponding to the order in which the calls were made.

D. SYSTEM INITIALIZATION

Because the Inner Traffic Controller's scheduler, GETWORK, can accommodate both normal calls and hardware interrupt jumps, the problem of system initialization is not difficult.

When SASS is first started at level-1, the Idle VP is running and the memory manager VP, which has the highest priority, is the first ready virtual processor in the ready list. All VP's available to the Traffic Controller for level-2 scheduling are ready. Their IDLE_FLAG's and PREEMPT flags are set.

At level-2, all VP's are loaded with idle processes and all supervisor processes are ready.

The kernel stack segment of each process is initialized to appear as if it had been saved by a hardware Preempt interrupt (Figure 40).

All CPU registers and the supervisor stack pointer are stored on the stack. R15 is reserved as the kernel stack point; R14 contains the DBR. All other registers can be used to pass initial parameters to the process. The order in which these registers appear on the stack supports the FLZ/ASM block-move instructions.
Figure 40: Initialized Stack
The status block contains the current value of the stack pointer, R15, and the preempt interrupt return flag. This flag is set to indicate that the process has been saved by a preempt interrupt. The first three items on the stack: the process entry point, the initial process flag control word, and an interrupt identifier, are also initialized to support the action of a hardware interrupt.

To start-up the system, R14 (the DBR) is set to the Idle process DBR; the CPU Program counter is assigned the PREEMPT_ENTRY point in GETWORK; the CPU Flag Control Word (FCW) is initialized for the kernel domain; and the CPU is started. Because the Idle_VP is the lowest priority VP in the system, it will place itself back in the ready state and move the Memory Manager in the running state. The Memory Manager will execute an interrupt return because the interrupt return flag was set by system initialization. There will be no work for this kernel process so it will call WAIT to place itself in the waiting state. The next ready VP is idling, but since it's IDLE_FLAG and PREEMPT flag are set, GETWORK will select it. It too will execute an interrupt return, but because its PREEMPT flag is set, it will call TC_PREEMPT_HANDLER. This will cause the first ready process to be scheduled. Each time a supervisor process blocks itself, the next idle VP will be selected and the sequence will be repeated.
THE NAVAL POSTGRADUATE SCHOOL SECURE ARCHIVAL STORAGE SYSTEM. P-ETC(U)

MAR 81 L A COX, R R SCHELL, S L PERDUE

UNCLASSIFIED NPS52-81-001
The action described above is in accord with normal operation of the system. The only unique features of initialization are the entry point (PREEMP-ENTRY: in GETWORK) and the values in the initialized kernel stack.

The implementation presented in this thesis has been run on a Z8000 developmental module. System initialization has been tested and executes correctly. At the current level of implementation, no process multiplexing function is available. There is no provision for unlocking the APT after an initialized process has been loaded as a result, a call to the Traffic Controller (viz., ADVANCE or WAIT). In a process multiplexed environment this would cause a system deadlock. Once the process left the kernel domain with a locked APT, no process would be able to unlock it. The Traffic Controller must handle this system initialization problem.
Chapter XIV
CONCLUSION

The implementation presented in this thesis created a security kernel monitor that runs on the Z8000 Developmental Module. This monitor supports multiprogramming and process management in a distributed operating system. The process executes in a multiple virtual processor environment which is independent of the CPU configuration.

This monitor was designed specifically to support the Secure Archival Storage System (SASS) [2, 9, 5]. However, the implementation is based on a family of Operating Systems [7] designed with a primary goal of providing multilevel security of information. Although the monitor currently runs on a single microprocessor system, the implementation fully supports a multiprocessor design.

A. RECOMMENDATIONS

Because the Zilog MMU is not yet available for the Z8000 Developmental Module, it was necessary to simulate the segmentation hardware. As Reitz explained [12], this was accomplished by reserving a CPU register, R14, as a Descriptor Base Register (DBR) to provide a link to the loaded address space. When the MMU becomes available, this simulation must be removed. This can be done in two steps.
First, the addressing format must be translated to the segmented form. This requires no system redesign.

Second, the switching mechanism must be modified to accommodate to use the MMU. This can be done by modifying the SWAP_DBRE portion of GETWORK to multiplex the MMU_IMAGE onto the MMU hardware and this can be accomplished by changing about a dozen lines of the existing code.

B. FOLLOW ON WORK

Although the monitor appears to execute correctly, it has not been rigorously tested. Before higher levels of the system are added, it is essential that the monitor be highly reliable. Therefore a formal test and evaluation plan should be developed.

An automated system generation and initialization mechanism is also required if the monitor to be is a useful tool in the development of higher levels of the design.

Once the monitor has been proven reliable and can be loaded easily, work on the implementation of the Memory Manager kernel process and the remainder of the kernel can continue.
This section contains excerpts from a Naval Postgraduate School MS Thesis by J. T. Wells [20]. The origins of these excerpts are:

- **PART E**
  - IMPLEMENTATION OF SEGMENT MANAGEMENT FOR A SECURE ARCHIVAL STORAGE SYSTEM

INTRODUCTION

SEGMENT MANAGEMENT FUNCTIONS

- SEGMENT MANAGER
- NON-DISCRETIONARY SECURITY MODULE
- MEMORY MANAGER

SUMMARY

SEGMENT MANAGEMENT IMPLEMENTATION

CONCLUSIONS AND FOLLOW ON WORK

Minor changes have been made for integration into this report.
Chapter XVI

INTRODUCTION

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

SEGMENT MANAGEMENT FUNCTIONS

A. 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_KNOW, 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
- 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.

2. Database

The Known Segment Table (KST) is the database used to manage segments. The KST is described in its tabular form and PLZ/SIS structured representation in Figure 41. 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).
---Segment_

<table>
<thead>
<tr>
<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 [4 KST_REC]
KST REC Record [MM_Handle Array [3 Word]
Size Word
Access_Mode Byte
In_Core Byte
Class Long
M_Seg_No Short_Integer
Entry_Number Short_Integer]

Figure 41: Known Segment Table

- 159 -
The Mentor_Seg_Nr field is a number representing the segment number of a segment's parent or "mentor" segment. Its importance will be discussed shortly.

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

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

B. 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.

C. MEMORY MANAGER

1. Function

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

The Distributed Memory Manager Module creates the following extended instruction set: MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. The instructions form the mechanism of communication between the Segment Manager of a process and a memory manager process (where the actual memory management functions are performed). The Memory Manager Process instruction set corresponds one to one with that of the Distributed Memory Manager; the set consists of: CREATE_ENTRY, DELETE_ENTRY, ACTIVATE, DEACTIVATE, SWAP_IN, and SWAP_OUT. The basic functions performed by the Memory manager are allocation/deallocation of global and local memory and of secondary storage, and segment transfers from local to global memory (and vice-versa) and from secondary storage to main memory (and vice-versa).

2. Databases

A detailed and descriptive discussion of the Memory Manager databases is presented in the work of Gary and Moore [5], 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 42) 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 regi-
ters of the hardware NMU). The NMU Image is an exact image of the NMU. 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 Blks_U Sed 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 (Figure 43). It is an array of memory manager VP_ID's (NM_VP_ID) indexed by CPU number. This table enables a signalling process to identify the appropriate memory manager process (virtual processor) to signal.
<table>
<thead>
<tr>
<th>Segment No.</th>
<th>Blocks Used</th>
<th>Max Avail Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MMU Image**
- Array [MAX_DBR_No MMU]

**MMU Record**
- SDR Array [No_Seg_Desc_Reg
- Seg_Desc_Reg]
  - Blks Used Word
  - Max_Elbs Word

**Seg_Desc_Reg**
- Record [Base_Addr Address
  - Limit Byte
  - Attributes Byte]

**Address**
- Word

Figure 42: Memory Management Unit Image

- 165 -
D. **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 the following chapter.
Chapter XVIII

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 [12] provided a demonstration of the operation of the Inner Traffic Controller primitives SIGNAL and WAIT (for interprocess communication). Integral to this demonstration was the correct performance of the Inner Traffic Controller VP scheduling mechanism and a "stub" of the Traffic Controller and its process scheduling mechanism (the TC support and use of the
mechanism of eventcounts and sequencers was not a part of the demonstration). The Segment management demonstration (hereafter referred to as "Seg_Mgr.Demo") was "built on top of" Reitz' ITC synchronization primitive demonstration (hereafter referred to as "Sync. Demo"). Thus, an immediate issue was to resolve the feasibility of adding on to Sync.Demo and also to refine the present design of the Sync. Demo to facilitate its integration into the Seg_Mgr.Demo. One aspect of this effort was in resolving the problem of how to pass (i.e., in interprocess communication) a larger message.

1. INTERPROCESS MESSAGES

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

Another alternative considered was that of a shared "Mailbox" segment with an associated eventcount acted on by the Kernel Inner Traffic Controller primitives TICKET, ADVANCE and AWAIT. A design for using this concept in the supervisor ring is provided by Parks [9]. This alternative was not deeply considered since these primitives are not included in the current Inner Traffic Controller.

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

2. **Structures as Arguments**

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

3. **Reentrant Code**

The issue of code reentrancy was addressed at the assembly language level through the use of a stack segment and registers for storage of local variables. PLZ/SYS (high level language) does not address reentrant procedures and thus the segment management high level code is not automatically reentrant. The problem of reentrancy can be seen by
looking at a shared procedure that is not reentrant; such a procedure has storage for its variables allocated statically in memory. Suppose a procedure (e.g., in the Kernel) can be activated by more than one process. While the procedure is executing in one process, a process switch occurs (e.g., to wait for a disk transfer) and its execution is suspended. The second process is activated, and while it is running it invokes the procedure. While the procedure is executing for the second process it uses the same storage space for variables as it did when executing for the first process. Eventually, it relinquishes the processor. However, when the procedure resumes its execution for the first process, the variable values that were in use by it originally have been changed during its execution in the second process. Thus, incorrect results are now inevitable.

4. **Process Structure of the Memory Manager**

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

Another key issue was that of the per process Segment Manager database, the KST. Since each process has its own KST, it cannot be linked to the (shared) segment manager procedures. To implement the KST as a per process database, it was convenient to establish, by convention, a KST segment number that is consistent from process to process. That segment in each process is the KST segment for that process. Implementation is then accomplished by using the segment number to construct a pointer to the base of the appropriate KST. It is then easy to calculate an appropriate offset to index any desired entry in the KST data.

6. **DBR Handle**

In Peitz'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 MHU_IMAGE record containing the list of descriptors for segments in the process address space. Gary and Moore [5] reference a "DBR_NO" that is essentially a handle used within the memory manager as an index within the MHU_IMAGE to a particular MHU record. The base address of the MHU 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.

- 172 -
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 [2]. 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_KNOW--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/ASM listings for the Segment Manager are in Appendix H.
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 [5] 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; MAKEKNOWN will accomplish that.
2. **Delete a Segment**

The function that deletes a segment (i.e., deletes a segment from SASS) is `DELETE_SEGMENT`. Validation of parameters and security checks are performed here similar to (but fewer than) the `CREATE_SEGMENT` checks. The distributed memory manager is then called to cause IPC with the memory manager process, where segment deletion is realized through secondary storage deallocation and alias table entry deletions. `DELETE_SEGMENT` is passed as arguments: (1) `Mentor_Seg_No` and (2) `Entry_No`. Conversion of the `Mentor_Seg_No` to a KST index is accomplished first. The pointer to the base of the KST is located and returned, as before. The mentor segment is checked to ensure it is known, again, by verifying that its own `M_SEG_No` (mentor segment number) entry in the KST is not the `NULL_SEG`. The process classification is obtained from the TC and checked (by a call to `CLASS_EQ`) to ensure it is equal to the mentor segment classification, since deleting an entry requires write access to the alias table. If all checks are satisfactory, then the mentor segment's `MM_Handle` pointer is derived. This pointer and the mentor segment alias table entry number are passed in a call to the distributed memory manager procedure `MM_DELETE_ENTRY`. It then performs IPC with the memory manager process where segment deletion is accomplished and a success code is returned as before.
3. **Make a Segment Known**

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

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

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

During the search of the KST, the M_SEG_No field is also checked to see if it contains the NULL_SEG entry (this implies that the segment number associated with the record is "available"). The first time this is noted, the index is saved. Note the first available index is saved since it is desired to assign segment numbers at the "top" of the KST to keep it dense there. When the search does not find that the segment is already known, the index for the available segment number is retrieved and converted to segment number by adding to it the number of kernel segments. If this index is the NULL_SEG entry, then there is no segment number available. In this event, the success code is set to NO_SEG_AVAIL, the segment number is assigned NULL_SEG, and access allowed is set to NULL_ACCESS (this is the third case mentioned). If the index is not equal to NULL_SEG and conversion to segment number has occurred then the Traffic Controller is called to provide the DBR_No (descriptor base register number) for the current process. The DBR_No is used by the memory manager process as an index in the MNU_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 $0$'s 0-9). Thus a check is made to ensure
that the segment number is not less than the number of Kernel segments; otherwise an error code is returned. Next, the segment number is checked to ensure that it is not larger than the maximum segment number allowable (if so, an error code is returned). If all checks are satisfactory, then the segment's MM_Handle pointer and the process DBR_No are obtained (as discussed before) and passed in a call to the MM_Deactivate procedure. It calls the memory manager process procedure DEACTIVATE which removes or updates (as appropriate) the entries in the local and global AST's.

5. **Swap a Segment In**

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

6. **Swap a Segment Out**

The function that swaps a segment from main memory to secondary storage is SM_SWAP_OUT. It is passed the segment number of the segment to be swapped out as an argument and returns a success code. The behavior of SM_SWAP_OUT is exactly analogous to that of SM_SWAP_IN except that the segment's KST IN_CORE entry is updated to reflect that the segment has been removed from main memory for this process (i.e., the new entry is "false").

C. **Non-Discretionary Security Module**

The Non-Discretionary Security Module implements the non-discretionary security policy for the SASS. The NDS module contains two procedures: CLASS_EQ and CLASS_GE; both compare two labels (classifications) and determine if their relationship meets that of the procedure's name (i.e., equal, or greater than or equal). Although the type of checks being made are, in fact, compatibility checks, Simple Security Condition checks, etc, the NDS Module does not recognize or need to recognize this. It simply uses an algorithm to determine if classification #1 = classification #2
or if classification #1 > classification #2, as appropriate. It then returns a condition code of true or false in accordance with the particular case. The earlier discussion of label comparison in accordance with a partially ordered lattice structure is relevant in discussing the NDS Module's algorithm. Consider the same "totally ordered" relationship $TS > S > C > U$ of levels and the "disjoint" relationship $Cy \mid Nu \mid Nu_1 \%$ 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:

$Cy \ OR \ N$

$(0000 \ 0000 \ 0000 \ 0010) \ OR \ (0000 \ 0000 \ 0000 \ 0010)$

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

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

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

Clearly, the answer is yes.
(2) Compare category #1 with category #2 -- is

(0000 0000 0000 0011) a superset of

(0000 0000 0000 0010), or more clearly

is the latter a subset of the former?

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

CyN OR N :

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

= 0000 0000 0000 0011 = category #1.

This means, in this example, that category #2 is a sub-
set (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 catego-
ry. Appendix I is the 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 >= level #2 (determine this by simple numerical comparison of values) and (2) category #2 subset category #1 (determine this by performing a logical OR with the categories and comparing the result to category #1 -- if they are equal then category #2 is a subset of category #1).

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

D. DISTRIBUTED MEMORY MANAGER MODULE

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

1. **Description of Procedures**

   Each procedure is invoked (and returns) on a one to one
   basis with a corresponding procedure in the Segment Manager.
   For example, CREATE_SEGMENT invokes MM_CREATE_ENTRY which
   signals the CREATE_ENTRY procedure in the Memory Manager
   Process Module. Associated with each procedure is an IPC
   message "frame" to describe the unique format of the con-
   tents of the message to be signalled to the memory manager
   process. Similarly, there must be a message "frame" for re-
   turn 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 en-
   sure that it is of length 16 bytes. The purpose of this
   frame is to provide an overlay onto the actual message array
   to be signalled and to facilitate loading the arguments into
the message array. This is accomplished by having a pointer of the type that points to the frame but by converting its address so that it actually points to the base of the message array. Consider these lines of PLZ/SYS code:

CE_MSGPTR := CE_PTR COM_MSGPTR
CE_MSGPTR..CREATE_CODE := CREATE_ENTRY_CODE

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

2. Interprocess Communication

The final arrangements and actual performance of IPC is completed by the internal procedure PERFORM_IPC. By locating the identity of the current physical processor (CPU) and using that identity to index into the MM_CPU_TABLE, the VP_ID of the current memory manager is resolved, so that the memory manager process dedicated to this physical processor is signalled. The call to K_LOCK is, in fact, a disguised call to the SPIN_LOCK procedure (since K_LOCK calls SPIN_LOCK). K_LOCK represents an ultimate (as yet unimplemented) goal of
a kernel locking (wait-lock) system. In any event, the G_AST lock must be set prior to signalling the memory manager process. After SIGNAL has been called, a call is made to WAIT with the pointer to the message array as the argument. The synchronization cycle that results is: (1) PERFORM_IPC calls the ITC procedure SIGNAL with the memory manager VP_ID and message array pointer as arguments; PERFORM_IPC then calls WAIT with the message array as the argument, (2) SIGNAL causes the message array to be copied into the message queue (in the VPT) of the appropriate VP_ID, (3) ultimately, the signalled VP is scheduled; it had previously called WAIT, passing a pointer to its own local message array; the action of WAIT is to copy the message from the VPT to the signalled process' local message array; there it is interpreted by the memory manager process main procedure and the appropriate procedure is called for action (e.g., CREATE_ENTRY), (4) when action is completed the memory manager process fills its local message array with the appropriate return message and calls SIGNAL with a pointer to the message and the original signalling process' VP_ID as arguments, (5) SIGNAL causes the memory manager process' message to be copied into the VPT message queue for the appropriate VP_ID, (6) that VP is eventually scheduled and through the action of WAIT has the return message copied from its message queue in the VPT to its local message array; WAIT then returns to PERFORM_IPC. The G_AST lock is unlocked and
PERFORM_IPC returns to the appropriate distributed memory manager procedure.

The last procedure in the distributed memory manager is \texttt{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 \texttt{SWAP\_VDBR} (remember that presently the Inner Traffic Controller deals with the DBR as the address of the appropriate MMU record in the MMU\_IMAGE while the Traffic Controller uses DBR as a DBR number which indexes to the appropriate MMU record).

E. \textbf{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 XII
CONCLUSIONS AND FOLLOW ON WORK

The implementation of segment management for the security kernel of a secure archival storage system has been presented. The implementation was completed on Zilog's Z8002 sixteen bit nonsegmented microprocessor. Segmentation hardware (Zilog's Z8010 Memory Management Unit) was not available, therefore it was simulated in software as described by Reitz [12]. 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 [5], 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 J. Prior to this implementation, formal testing of the segment management implementation herein and the monitor implemented by Reitz [12] should be completed.
This section contains excerpts from a Naval Postgraduate School MS Thesis by A. R. Strickler [19]. The origins of these excerpts are:

INTRODUCTION from Chapter I
IMPLEMENTATION ISSUES from Chapter III
PROCESS MANAGEMENT IMPLEMENTATION from Chapter IV
CONCLUSION from Chapter V

Minor changes have been made for integration into this report.
Chapter IX

INTRODUCTION

This thesis addresses the implementation of process management functions for the Secure Archival Storage System or SASS. This system is designed to provide multilevel secure access to information stored for a network of possibly dissimilar host computer systems and the controlled sharing of data amongst authorized users of the SASS. Effective process management is essential to insure efficient use and control of the system.

The major accomplishments of this thesis effort include the provisions for efficient process creation and management. These functions are provided through the establishment of a system Traffic Controller and the creation of a virtual interrupt structure. An effective mechanism for inter-process communication and synchronization is realized through an Event Manager that makes use of uniquely identified segments supported by eventcount and sequencer primitives. A hardware controlled two domain operational environment is created with the necessary interfacing between domains provided by a software "gate" mechanism. Additional support is provided through considerable work in the area of database initialization and a technique for limited dynamic memory allocation.
This implementation was completed on the commercial AMC Am96/4116 MonoBoard Computer with a standard Multibus interface.
Chapter XXI

IMPLEMENTATION ISSUES

Issues bearing on the implementation of process management and refinements made to existing modules are presented in this chapter. Process management for the SASS was provided through the implementation of the Traffic Controller Module, the Event Manager Module, the Distributed Memory Manager Module, and a Gate Keeper Stub (system trap). Additionally, since a demonstration/testbed was integral to the testing and verification of the implementation, it was necessary to complete other supportive tasks. These supportive tasks included limited Kernel database initialization, revised preempt interrupt handling mechanisms, Idle process definition and structure, and additional refinements to existing modules.

A. DATABASE INITIALIZATION

Previous work on SASS has relied on statically built databases, which proved to be sufficient for demonstration of a single processor, single host supported system. In the current demonstration, multiple hosts are simulated, and the Kernel data structures have been refined to represent a multiprocessor environment. Since a multiprocessor system was
unavailable at the time of this demonstration, several "runs" were made and traced, using different logical CPU numbers, to show the correctness of this structure. Due to this multiprocessor representation and simulation of multiple hosts, the use of statically built Kernel databases was no longer convenient. Therefore, it became necessary to provide initialization routines for the dynamic creation of those Kernel databases required for this implementation. While it was not the intent of this effort to implement system initialization, care was taken in the writing of these initializing routines so that they might be utilized in the system initialization implementation with, hopefully, minimal refinement. Database initialization was restricted to those databases existing in the Inner Traffic Controller and the Traffic Controller. Limited elements of the Known Segment Table (KST) and Global Active Segment Table (G_AST) were also created for demonstration purposes.

1. **Inner Traffic Controller Initialization**

A "Bootstrap Loader" Module, which logically exists at a higher level of abstraction within the Kernel, was created to initialize the databases of the Inner Traffic Controller. This initialization includes the creation of: 1) the Processor Data Segment (PRDS), 2) an MMU Map, 3) Kernel domain stack segments for Kernel processes, 4) allocation and updating of MMU entries for Kernel processes, and 5) Virtual Processor Table (VPT) entries.
The PRDS was loaded with constant values that specify the physical CPU ID, logical CPU ID, and number of VP's allocated to the CPU. A design decision was made to allocate logical CPU ID's in increments of two (beginning with zero) so that they could be used to directly access lists indexed by CPU number. The MMU map, constructed as a "byte" map, was created to specify allocated and free MMU Image entries.

A separate procedure, CREATE_STACK, was created to establish the initial Kernel domain stack conditions for Kernel processes. A discussion and diagram of these initial stack conditions is presented in the next section. ALLOCATE_MMU checks the MMU Map and allocates the next available MMU entry to the process being created. The PRDS is inserted in the allocated MMU entry and the DBR number is returned to the calling procedure. The DBR number (handle) is merely the offset of the DBR in the MMU Image. Since the ITC deals with an address rather than a handle, a procedure, GET_DBR_ADDR, was created to convert this offset into a physical address. UPDATE_MMU_IMAGE is the procedure which creates or modifies MMU Image entries. UPDATE_MMU_IMAGE accepts as arguments the DBR number, segment number, segment attributes, and segment limits. To facilitate process switching and control, various process segments must possess the same segment number system wide. This is accomplished during initialization through the use of the UPDATE_MMU_IMAGE procedure. In the ITC, these segments in-

- 200 -
clude the PRDS (segment number zero) and the Kernel stack segment (segment number one).

The final task required in ITC initialization is the creation of the VPT. The VPT header is initialized with the "running" and "ready" lists pointers set to a 'nil' state, and the "free" list pointer set to the first entry in the message table. Virtual Processor entries are inserted in the main body of the VPT by the UPDATE_VP_TABLE procedure. Entries are first made for the VP's permanently bound to the Memory Manager and Idle processes. The VP bound to the MM process is given a priority of 2 (highest), and the VP bound to the Idle process is given a priority of 0 (lowest). The External VP ID for both of these VP's is set to "nil" as they are not visible to the Traffic Controller. The remaining VP's allocated to the CPU (viz., TC visible VP's) are then entered in the VPT with a priority of 1 (intermediate), and their "idle" and "preempt" flags are set. The preempt flag is set for these TC visible VP's to insure proper scheduling by the Traffic Controller. The DBR for these remaining VP's is initialized with the Idle process DBR. A discussion of "idle" processes and VP's will be provided later in this chapter. The External VP ID for each TC visible VP is merely the offset of the next available entry in the EXTERNAL VP LIST. This External VP ID is entered in the VPT, and the corresponding VP ID (viz., VPT Entry #) is entered in the EXTERNAL VP LIST.
Once these VPT entries have been made, it is necessary to set the state of each VP to "ready" and thread them (by priority) into the appropriate ready list. A VPT threading mechanism was provided by Reitz [12] in procedure MAKE READY. However, it was desired to have a more general threading mechanism that could be used for other lists. Procedure LIST_INSERT was created to provide this general threading mechanism. LIST_INSERT is logically a "library" function that exists at the lowest level of abstraction in the Kernel. This function threads an object into a list (specified by the caller) in order of priority, and then sets its state as specified by the calling parameters.

Once the "Bootstrap Loader" has completed ITC initialization, it passes control to the ITC GETWORK procedure to begin VP scheduling.

2. Traffic Controller Initialization

The initialization routines for the TC include TC_INIT, CREATE_PROCESS, and CREATE_KST. These routines are called from the Memory Manager process. The MM process was chosen to initiate these routines as it is bound to the highest priority VP and will begin running immediately after the Inner Traffic Controller is initialized. Procedure MM_ALLOCATE was written to allocate memory space for data structures during initialization (viz., Kernel stacks, user stacks, and KST's). Memory space is allocated in blocks of
100 (hex) bytes. MM_ALLOCATE is merely a stub of the memory allocating procedure designed by McCre and Gary [5].

It was necessary to pass long lists of arguments to the TC for initialization purposes. To aid in this passing of parameters, a data structure template was used. This template was created by declaring the parameters as a data structure in both the sending and receiving procedures, and then imaging this structure at absolute address zero. The process' stack pointer was then decremented by the size of the parameter data structure, and the parameters were loaded into this data structure indexed by the stack pointer. This template made it very easy to send and receive long argument lists using the process' stack segment.

TC_INIT initializes the APT header and virtual interrupt vector (discussed later). Each element of the running list is marked "idle", the ready and blocked lists are set to "nil", and the number of VP's and first VP for each CPU are entered in the VP table. The address of the virtual preempt handler is then passed to the ITC procedure CREATE_INT_VEC for insertion in the virtual interrupt vector.

CREATE_PROCESS initializes user processes and creates entries in the APT. ALLOCATE_MNU is called to acquire a DBR number, and an APT entry is created with the process descriptors (viz., parameters). The process is then declared "ready" and threaded into the appropriate ready list by calling the threading function, LIST_INSERT. A user stack
is allocated and UPDATE_MNU_IMAGE is called to include the user stack in the MNU as segment number three. The user stack contains no information or user process initialization parameters (viz., execution point and address space) as all processes are initialized and begin execution from the Kernel domain. Next, a Kernel domain stack is allocated and included in the MNU Image. A design decision was made to initialize the Kernel stacks for user processes with the same structure as the Kernel process' stacks. The rationale for this decision is presented in the next section. As a result of this decision, it became possible to use the CREATE_STACK procedure in building Kernel domain stacks for both Kernel and user processes. CREATE_STACK was therefore used as a library function and placed in the library module with LIST-INSERT.

Finally, a Known Segment Table (KST) stub is created to provide a means of demonstrating the mechanism provided by the eventcounts and sequencers for interprocess communication (IPC) and mutual exclusion. Space for the process' KST is created by calling NALLOCATE. The KST is then included in the process' address space, as segment number two, by UPDATE_MNU_IMAGE. Initial entries are made in the Known Segment Table by procedure CREATE_KST. CREATE_KST makes an entry in the KST for the "root" and marks the remaining KST entries as "available." The Unique_ID portion of the root's handle (viz., upper two words) is initialized as -1 (for
convenience) and the G_AST entry number portion of the handle (viz., lowest word) is initialized with zero.

3. **Additional Initialization Requirements**

As already mentioned, the Memory Manager Process prepares the arguments utilized by TC_INIT, CREATE_PROCESS, and CREATE_KST for TC initialization and user process creation. Additionally, the MM process creates a Global Active Segment Table (G_AST) stub utilized for demonstration of event data management. The G_AST stub is declared in a separate module (viz., the DEMO_DATABASE Module) with the format prescribed by Moore and Gary [5]. However, the only fields initialized and utilized by this implementation are UNIQUE_ID, SEQUENCER, INSTANCE 1, and INSTANCE 2. The eventcounts and sequencer fields are initialized as zero whenever an entry is created in the G_AST. The UNIQUE_ID is created just to support this demonstration and does not reflect the segment's unique identifier as specified by Moore and Gary [5]. In this demonstration, UNIQUE_ID is built with the parameters passed to MM_ACTIVATE. The first word in UNIQUE_ID is the G_AST entry number of the segment's parent, and the second word is the segment's entry number into the alias table. The UNIQUE_ID together with the offset of the segment's entry in the G_AST comprise the segment HANDLE maintained in the KST. The first entry in the G_AST is reserved for the root, and is initialized with an Unique_ID of minus one.
(-1). It should be noted that any call to \texttt{MM\_ACTIVATE} for a segment already possessing an entry in the \texttt{G\_AST} will not effect any changes to that entry. This is to insure that a single \texttt{G\_AST} entry exists for every segment as specified by Moore and Gary [5].

B. PREEMPT INTERRUPTS

Various refinements were made in the handling of both physical (hardware) and virtual (software) preempt interrupts. A hardware preempt is a non-vectored interrupt that invokes the virtual processor scheduling mechanism (viz., \texttt{ITC\_GETWORK}). A virtual preempt is a software vectored interrupt that invokes the user process scheduling mechanism (viz., \texttt{TC\_GETWORK}). This implementation provides the notion of a virtual interrupt that closely mirrors the behavior of a hardware interrupt. In particular, there are similar constructs for initialization of a handler, invocation of a handler, masking of interrupts, and return from a handler. As with most hardware interrupts, a virtual interrupt can occur only at the completion of execution for an "instruction," where each kernel entry and exit for a process delimit a single "virtual instruction."
1. **Physical Preempt Handler**

The physical preempt handler resides in the virtual processor manager (viz., Inner Traffic Controller). The functions it perform are: 1) save the execution point, 2) invoke ITC GETWORK, 3) check for virtual preempt interrupts, 4) restore the execution point, and 5) return control via the IRET instruction. Reitz [12] included the hardware preempt handler in ITC GETWORK by establishing two entry points and two exit points, one for a regular call to GETWORK and another for the preempt interrupt. He had a separate procedure, TEST_PREEMPT, that was used to check for the occurrence of virtual preempt interrupts. This structure works nicely, but it requires some means of determining how GETWORK was invoked so that the proper exiting mechanism is used. This was resolved by incorporating a preempt interrupt flag in the status register block of every process' Kernel domain stack segment. A design decision was made to restructure the hardware preempt handler into a single and separate procedure, PHYS_PREEMPT_HANDLER. This allowed ITC GETWORK to have a single entry and exit point, and it did away with the necessity of maintaining a preempt interrupt flag in the process stacks. PHYS_PREEMPT_HANDLER was constructed from the preempt handling code in GETWORK and procedure TEST_PREEMPT. TEST_PREEMPT was deleted from the ITC as its functions were performed by PHYS_PREEMPT_HANDLER.
A further refinement was made to the hardware preempt handler dealing with the method by which the virtual preempt handler was invoked. Reitz [12] invoked the virtual preempt handler from TEST_PREAMPT by means of the "call" instruction. Since the virtual preempt handler logically exists at a higher level of abstraction than the ITC, this invocation violated our notion of only allowing "calls" to lower or equal abstraction levels. However, this deviation was necessitated by the absence of a virtual interrupt structure. This problem was alleviated by creating a virtual interrupt vector in the ITC that is used in the same way as the hardware interrupt vector. The virtual preempt was given a virtual interrupt number (zero). The virtual interrupt handler is then invoked by means of a "jump" through the virtual interrupt vector for virtual interrupt number 0. This invocation occurs in the same manner that the handlers for hardware interrupts are invoked. The virtual interrupt vector is created by procedure CREATE_INT_VEC. CREATE_INT_VEC accepts as arguments a virtual interrupt number and the address of the interrupt handler. The creation of the virtual preempt entry in the virtual interrupt vector is accomplished at the time of the Traffic Controller initialization by TC_INIT.
2. **Virtual Preempt Handler**

The virtual preempt handler (VIRT_PREEMPT_HANDLER) resides in the user process manager (viz., the Traffic Controller). The functions performed by VIRT_PREEMPT_HANDLER are: 1) determine the VP ID of the virtual processor being preempted, 2) invoke the process scheduling mechanism (viz., TC_GETWORK), and 3) return control via a virtual interrupt return. The correct VP ID is obtained by calling RUNNING_VP in the ITC. The Active Process Table is then locked, and the state of the process running on that VP is changed to "ready." At this time, process scheduling is effected by calling TC_GETWORK. Once process scheduling is completed, the APT is unlocked and control is returned via a virtual interrupt return. This virtual interrupt return is merely a jump to the PREEMPT_RET label in the hardware preempt handler (This jump emulates the action of the IRET instruction for a hardware interrupt return). This label is the point at which the virtual preempt interrupts are unmasked.

All Kernel processes are initialized to appear as though they are returning from a hardware preempt interrupt. All user processes initially appear to be returning from a virtual preempt interrupt. Therefore, the initial conditions of a process' Kernel domain stack is largely influenced by the stack manipulation of the preempt handlers. Figure 44 illustrates the initial Kernel domain stack structure for all system processes.
Figure 44: Initial Process Stack
The initial Kernel Flag Control Word (FCW) value is "5000", indicating non-segmented code, system mode of operation, non-vectored interrupts masked, and vectored interrupts enabled. The Current Stack Pointer value is set to the first entry in the stack (viz., SP). The IRET Frame is the portion of the Kernel stack affected by the IRET instruction. The first element, Interrupt ID (set to "FFFF") is merely popped off of the stack and discarded. The next element, Initial FCW, is popped and placed in the system Flag Control Word. Initial FCW is set to "5000" for Kernel processes and "1800" (indicating normal mode with all interrupts enabled) for user processes. The final element of the IRET frame, Initial IC is popped off of the stack and placed in the program counter (PC) register. This value is initialized as the entry address of the process in question.

The "register" entries on the stack represent the initial register contents for the process at the beginning of its execution. Since the Kernel processes (viz., MM and Idle) do not require any specific initial register states, their entries reflect the register contents at the time of stack creation. Initial register conditions are used to provide initial "parameters" required by the user processes. This will depend largely upon the parameter passing conventions of the implementation language. The means for register initialization was provided through CREATE_PROCESS; however, the only initial register condition used for the user
processes in this demonstration was register #13. Register #13 was used to pass the user ID/Host number of the process created. This value is utilized by the user process in activating the segment used for inter-process communication between a Host's File manager and I/O processes. Another logical parameter passed to the user processes is the root segment number. This did not require a register for passing in the demonstration as it is known to be the first entry in the KST for all processes. The N_S_P entry on the stack represents the initial value of the normal stack pointer. For user processes, this value is obtained when the Supervisor domain stack for that process is created. For Kernel processes, this value is set to "FFFF" since they execute solely in the Kernel domain and have no Supervisor domain stack. The Preempt Return Point specifies the address where control will be passed once the process' VP is scheduled and the "return" from ITC GETORK is executed. For Kernel processes, this is the point within the hardware preempt handler where the virtual processor table is unlocked. For user processes, this is the point within the virtual preempt handler where the Active Process Table is unlocked.

It is important to note that if the APT was not unlocked when a user process began its initial execution, the system would become deadlocked and no further process scheduling could occur. It should be further noted that the initial stack conditions for user processes do not reflect a valid...
history of execution. The "normal" history of a user process returning from ITC_GETWORK after a virtual preempt interrupt would reflect the passing of control through SWAP_VDBR and TC_GETWORK to the point in the virtual preempt handler where the APT is unlocked. Another "possible" history could reflect the occurrence of a hardware preempt interrupt at the point in the virtual preempt handler where the APT is unlocked. Such a history would be depicted by replacing the current top of the stack with the return point into the hardware preempt handler (viz., at the point of virtual preempt interrupt unmasking) and an additional hardware preempt interrupt frame whose IC value in the IRET frame is the point in the virtual preempt handler where the APT is unlocked. The current initial stack condition for user processes was chosen for its ease of understanding and its clear depiction of the fact that the structure of a Kernel domain stack is the same for both Kernel and user processes.

C. IDLE PROCESSES

In the SASS design, there logically exists a Kernel domain "Idle" process for every physical processor in the system and a Supervisor domain "Idle" process for every "TC visible" virtual processor in the system. These processes are necessary to insure that both the VP scheduler (viz., ITC_GETWORK) and the process scheduler (TC_GETWORK) will always
have some object to schedule, hence precluding any CPU or VP from ever having an undefined execution point. Since the Kernel domain Idle process performs no useful work, it could be included within the ITC by means of an infinite looping mechanism. The Kernel Idle process was maintained separately, however, as it is hoped that future work on SASS will provide this Idle process with some constructive purpose (e.g., performing maintenance diagnostics).

The Supervisor domain Idle processes (hereafter referred to as TC Idle processes) are scheduled (bound) on VP's when there are no user processes awaiting scheduling. Since a TC Idle process performs no user constructive work, we do not want any VP executing a TC Idle process to be bound to a physical processor. In other words, a VP bound to a TC Idle process assumes the lowest system priority (represented by the "idle flag"). Therefore, any such VP will have its idle flag set and will not be scheduled unless it receives a virtual preempt interrupt. Such an interrupt will allow the VP to be rescheduled by the Traffic Controller. It should be obvious, at this point, that a TC Idle process will never actually begin execution on a real processor. This knowledge allowed a design decision to be made to only simulate the existence of TC Idle processes. At the TC level, this was accomplished by a constant value, IDLE_PROC, that was used as a process ID in the APT running list, thus precluding the necessity of any "Idle" entries in the APT. At the
ITC level, any VP marked "Idle" (viz., the idle flag set) was given the DBR number (viz., address space) of the Kernel Idle process solely to provide the use of a Kernel domain stack for rescheduling of the VP.

D. ADDITIONAL KERNEL REFINEMENTS

In addition to those already discussed, several other refinements to existing Kernel modules were effected in this implementation. One of these refinements deals with the way virtual processors are identified by the Traffic Controller. In the current implementation, all TC visible virtual processors are given an External VP ID which corresponds to its entry number in an External VP List. This required a modification to the ITC procedure RUNNING_VP. The benefits derived from this refinement included the ability to directly access the External VP ID in the Virtual Processor Table vice the requirement of a runtime division to compute its value and the ability to use the External VP ID as an index into the TC running list.

Refinements were also made to the existing Memory Manager, File Manager and IO process stubs used for demonstration purposes. These refinements were largely associated with the eventcount and sequencer mechanisms utilized in this implementation. The current status of these processes is provided in this report.

- 215 -
The remaining refinements deal largely with the MMU Image. In Moore and Gary's [5] design, the MMU Image was managed by the Memory Manager process. This was largely because the MMU Image is a processor local database and would seem well suited for management by the non-distributed kernel. In fact, the MMU Image is utilized mainly by the ITC for the multiplexing of process address spaces. Therefore, in the current design, the MMU Images are maintained by the Inner Traffic Controller. However, the MMU header proposed by Moore and Gary (viz., the BLOCKS_USED and MAXIMUM_AVAILABLE_BLOCKS fields) was retained in the Memory Manager as it is used strictly in the management of a process' virtual core and is not associated with the hardware MMU.

In Wells' design [20], the Traffic Controller used the linear ordering of the DBR entries in the MMU Image as the DBR handle (viz., 1,2,3...). This required a runtime division operation to compute the DBR number, and a runtime multiplication operation, by MM_GET_DBR_VALUE, to recompute the DBR address for use by the ITC. In the current design, the offset of the DBR entry in the MMU Image (obtained at the time of MMU allocation) is used as the DBR handle in the Traffic Controller. Furthermore, SWAP_VDBR was refined to accept a DBR handle rather than a DBR address to preclude the necessity of the Traffic Controller having to deal with MMU addresses. DBR addresses are computed only within the

- 216 -
ITC (viz., by procedure GET_DBR_ADDR) by adding the value of the DBR handle to the base address of the MMU Image. Since DBR addresses are now used solely within the ITC, procedure MM_GET_DBR_VALUE was no longer needed and was deleted from the Memory Manager.

E. SUMMARY

The primary issues addressed in this thesis effort have been presented in this chapter. Aside from the process management functions, this description included a mechanism for limited Kernel database initialization, a revised preempt interrupt handling mechanism, the creation of a virtual interrupt structure, a definition of "idle" processes and their structure, and a discussion of the minor refinements effected in existing SASS modules. A detailed description of the implementation of process management functions for the SASS is presented in the next chapter.
Chapter XXIII

PROCESS MANAGEMENT IMPLEMENTATION

The implementation of process management functions and a gate keeper stub (system trap) is presented in this chapter. The implementation is discussed in terms of the Event Manager, Traffic Controller, Distributed Memory Manager, User Gate, and Kernel Gate Keeper modules. A block diagram depicting the structure and interrelationships of these modules is presented in Figure 45. Support in developing the Z8000 machine code for this implementation was provided by a Zilog MCZ Developmental System operating under the RIO operating system. The Developmental System provided disk file management for a dual drive, hard sectored floppy disk, a line oriented text editor, a PLZ/ASM assembler, a linker and a loader that created an executable image of each Z8000 load module. An upload/download capability with the Am96/4116 MonoBoard computer was also provided. This capability, along with the general interfacing of the Am96/4116 into the SASS system, was accomplished in a concurrent thesis endeavor by Gary Baker. Baker's work relating to hardware initialization in SASS, will be published upon completion of his thesis work in June 1981.
Figure 45: Implementation Module Structure
A. EVENT MANAGER MODULE

The eventcount and sequencer primitives [11], which are system-wide objects, collectively comprise the event data of SASS. As mentioned earlier, this event data is tied directly to system segments and is stored in the Global Active Segment Table. There are two eventcounts and one sequencer for every segment in the system. These objects are identified to the Kernel in user calls by specification of a segment number. Once this segment number is identified by the Kernel, the segment's handle can be obtained from the process' Known Segment Table. The segment handle identifies the particular entry in the G_AST containing the event data desired.

The Event Manager module manages the event data within the system and provides the mechanism for interprocess communication between user processes. The Event Manager consists of six procedures. Four of these (Advance, Await, Read, and Ticket) represent the four user extended instructions provided by the Event Manager. The remaining two procedures provide internal computational support to include necessary security checking. The Event Manager is invoked solely by user processes, via the Gate Keeper, through utilization of the extended instruction set provided. For every Event Manager extended instruction invoked by a user process, the non-discretionary security is verified by com-
paring the security access classification of the process invoking the instruction with the classification of the object (segment) being accessed. Access to the user process' Known Segment Table is required by the module in order to ascertain the segment handle and security class for a given segment number. The PLZ/ASM assembly language listing for the Event Manager module is provided in Appendix A. A more detailed discussion of the procedures comprising the Event Manager follows.

1. **Support Procedures**

   The procedures GET_HANDLE and CONVERT_AND_VERIFY provide internal support for the Event Manager and are not visible to the user processes. Procedure CONVERT_AND_VERIFY is invoked by the four procedures representing the instruction set of the Event Manager. The input parameters to CONVERT_AND_VERIFY are a segment number and a requested mode of access (viz., read or write). CONVERT_AND_VERIFY returns a pointer to the segment's handle and a success code. Procedure GET_HANDLE is invoked solely by CONVERT_AND_VERIFY. The input parameter to GET_HANDLE is the segment number received as input by CONVERT_AND_VERIFY. GET_HANDLE returns a pointer to the segment's handle, a pointer to the segment's security classification, and a success code. A discussion of the functions provided by these support procedures follows.
Procedure GET_HANDLE translates the segment number, received as input, into a KST index number and verifies that the resulting index number is valid. Next the base address of the process' KST is obtained from procedure ITC_GET_SES_PTR. The KST index number is then converted into a KST offset value and added to the base address to obtain the appropriate KST entry pointer for the segment in question. A verification is then made to insure that the referenced segment is "known" to the process. If the segment is not known, an error message is returned to CONVERT_AND_VERIFY. Otherwise, a pointer to the segment's handle is obtained to identify the segment to the memory manager. A pointer to the segment's security class entry in the KST is also returned for use in appropriate security checks.

Procedure CONVERT_AND_VERIFY provides the necessary non-discretionary security verification for the extended instruction set of the Event Manager. Procedure GET_HANDLE is invoked for segment number verification and to obtain pointers to the segment's handle and security class. If GET_HANDLE returns with a successful verification, the process's security class is compared to the segment's security class to verify the mode of access requested. A request for "write" access causes invocation of the CLASS_EQ function in the Non-Discretionary Security Module to insure that the security classification of the process is equal to the classi-
fication of the eventcount or sequencer, which is the same as that of the segment. Otherwise, the CLASS_GE function is called to verify that the process has read access. If the appropriate security check is unsuccessful, an error code is returned by CONVERT_AND_VERIFY. Otherwise, the segment handle is returned along with a success code of "succeeded" indicating that the user process possesses the necessary security clearance to complete execution of the extended instruction.

2. **Read**

Procedure READ ascertains the current value of a user specified eventcount and returns its value to the caller. The input parameters to READ are a segment number and an instance (viz., an event number). CONVERT_AND_VERIFY is invoked with a "read" access request to obtain the segment's handle and necessary verification. "Read" access is sufficient for this operation as it only requires observation of the current eventcount value and performs no data modification. If verification is successful, procedure MM_READ_EVENTCOUNT is called to obtain the eventcount value.

3. **Ticket**

Procedure TICKET returns the current sequencer value for the segment specified by the user. CONVERT_AND_VERIFY is called with a request for write access to obtain verifica-
tion and the segment handle. Write access is required because once the sequencer value is read it must be incremented in anticipation of the next ticket request. Once verification is complete, MM_TICKET is invoked to obtain the sequencer value that is returned to the user process. It is noted that every call to TICKET for a particular segment number will return a unique and time ordered sequencer value. This is because the sequencer value may only be read within MM_TICKET while the G_AST is locked, thereby preventing simultaneous read operations. Furthermore, once the sequencer value is read it is incremented before the G_AST is unlocked.

4. **Await**

Procedure AWAIT allows a user process to block itself until some specified event has occurred. The parameters to AWAIT include a segment number and instance, which identify a particular event, and a user specified value which identifies a particular occurrence of the event. Verification of read access and a pointer to the segment's handle is obtained from procedure CONVERT_AND_VERIFY. Procedure TC_AWAIT is invoked to effect the actual waiting for the event occurrence. TC_AWAIT will not return to AWAIT until the requested event has occurred. It is noted that AWAIT makes no assumptions about the event value specified by the user. Therefore, the Kernel cannot guarantee that the event
specified by the user will ever occur; this is the responsibility of other cooperating user processes.

5. **Advance**

Procedure **ADVANCE** allows a user process to broadcast the occurrence of some event. This is accomplished by incrementing the value of the eventcount associated with the event that has occurred. The parameters to ADVANCE include a segment number and instance which identify a particular event. The calling process must have write access to the identified segment as modification of the eventcount is required. Verification of write access and a pointer to the segment's handle is obtained through procedure CONVERT_AND_VERIFY. Procedure TC_ADVANCE is invoked to perform the actual broadcasting of event occurrence.

B. **Traffic Controller Module**

The primary functions of the Traffic Controller module are user process scheduling and support of the inter-process communication mechanism. The Traffic Controller is invoked by the occurrence of a virtual preempt interrupt and by the Event Manager and the Segment Manager through the extended instruction set: TC_Advance, TC_Await, Process_Class, and Get_DBR_NUMBER. The Traffic Controller module is comprised of nine procedures. Four of these procedures represent the extended instruction set of the Traffic Controller. A de-
tailed discussion of six of the procedures contained in the Traffic Controller module is presented below. The remaining three procedures (viz., TC_INIT, CREATE_PROCESS, and CREATE_KST) were described in chapter three. The PLZ/ASM assembly language source code listings for the Traffic Controller module is provided in Appendix B.

1. **TC_GETWORK**

Procedure TC_GETWORK provides the mechanism for user process scheduling. The input parameters to TC_GETWORK are the VP ID of the virtual processor to which a process will be scheduled and the logical CPU number to which the virtual processor belongs. The determination of which process to schedule is made by a looping mechanism that finds the first "ready" process on the ready list associated with the current logical CPU number. Processes appear in the ready list by order of priority. This looping mechanism is required as both "running" and "ready" processes are maintained on the ready list. This ready list structure was chosen to simplify the algorithm provided in procedure TC_Advance. If a ready process is found, its state is changed to "running" and its process ID (viz., the APT entry number) is inserted in the running list entry associated with the current virtual processor. Procedure SWAP_VDBR is then invoked in the Inner Traffic Controller to effect the actual process switch. If a ready process was not found (viz., the ready
list was empty or comprised solely of "running processes"), then the running list entry associated with the current virtual processor is marked with the constant "Idle_Proc" and procedure IDLE is invoked in the Inner Traffic Controller.

2. **TC_AWAIT**

The primary function of TC_AWAIT is the determination of whether some user specified event has occurred. If the event has occurred, control is returned to the caller. Otherwise, the process is blocked and another process is scheduled. The input parameters to TC_AWAIT are a pointer to a segment handle, an instance (event number), and a user specified eventcount value. TC_AWAIT initially locks the Active Process Table and obtains the current value of the eventcount in question by calling procedure Mn_READ_EVENTCOUNT. The determination of event occurrence is made by comparing the user specified eventcount value with the current eventcount. If the user value is less than or equal to the current eventcount, the awaited event has occurred and control is returned to the caller. Otherwise, the awaited event has not yet occurred and the process must be blocked.

If the process is to be blocked, procedure RUNNING_VP is invoked to ascertain the VP ID of the virtual processor bound to the process. The process' ID (viz., APT entry number) is then read from the running list. The input parame-
ters to TC_AWAIT (viz., Handle, Instance, and Value) are then stored in the Event Data portion of the process' APT entry. The process is removed from its associated ready list by redirecting the appropriate linking threads (pointers). Once removed from the ready list, the process is threaded into the blocked list and its state changed to "blocked" by invocation of the library function LIST_INSERT. Procedure TC_GETWORK is then called to schedule another process for the current virtual processor.

3. **TC_ADVANCE**

The primary purpose of TC_ADVANCE is the broadcasting of some event occurrence. This entails incrementing the eventcount associated with the event, awakening all processes that are waiting for the event, and insuring proper scheduling order by generating any necessary virtual preempt interrupts. The high level design algorithm for TC_ADVANCE is provided in Figure 46. The input parameters to TC_ADVANCE are a pointer to a segment's handle and an instance (event number). Initially, TC_ADVANCE locks the APT to prevent the possibility of a race condition. The eventcount identified by the input parameters is then incremented by calling MM_ADVANCE. MM_ADVANCE returns the new value of the eventcount. Once the eventcount has been advanced, TC_ADVANCE awakens all processes awaiting this event occurrence. This is accomplished by checking all processes that are currently
in the blocked list. The process' HANDLE and INSTANCE entries are compared with the handle and instance identifying the current event. If they are the same, then the process is awaiting some occurrence of the current event. In such a case, the process' VALUE entry in the APT is compared with the current value of the eventcount. If the process' VALUE is less than or equal to the current eventcount value, the awaited event has occurred and the process is removed from the blocked list and threaded into the appropriate ready list by the library function LIST_INSERT.

Once the blocked list has been checked, it is necessary to reevaluate each ready list to insure that the highest priority processes are running. It is relatively simple to determine if a virtual preempt interrupt is necessary, however, it is considerably more difficult to determine which virtual processor should receive the virtual preempt. To assist in this evaluation, a "count" variable (number of preempts needed) is zeroed and a preempt vector is created on the Kernel stack with an entry for every virtual processor associated with the logical CPU being evaluated. Initially, every entry in the preempt vector is marked "true" indicating that its associated virtual processor is a candidate for preemption. Once the preempt vector is initialized, the first "n" processes on the ready list (where n equals the number of VP's associated with the current logical CPU) are checked for a determination of their state. If
TC_ADVANCE Procedure (HANDLE, INSTANCE)

Begin

! Get new eventcount !
COUNT := MR_ADVANCE (HANDLE, INSTANCE)

Call WAIT_LOCK (APT)

! Wake up processes !
PROCESS := BLOCKED_LIST_HEAD

Do while not end of BLOCKED_LIST
  If (PROCESS.HANDLE = HANDLE) and (PROCESS.INSTANCE = INSTANCE) and (PROCESS.COUNT <= COUNT)
    then
      Call LIST_INSERT(READY LIST)
    end if
  PROCESS := PROCESS.NEXT_PROCESS
end do

! Check all ready lists for preempts !
LOGICAL_CPU_NO := 1

Do while LOGICAL_CPU_NO <= #NR_CPU
  ! Initialize preempt vector !
  VP_ID := FIRST_VP(LOGICAL_CPU_NO)
  Do for LOOP := 1 to Nr_VP(LOGICAL_CPU_NO)
    RUNNING_LIST[VP_ID].PREEMPT := #TRUE
    VP_ID := VP_ID + 1
  end do

! Find preempt candidates !
CANDIDATES := 0
PROCESS := READY_LIST_HEAD(LOGICAL_CPU_NO)

Figure 46: TC_ADVANCE Algorithm

- 230 -
VP_ID := FIRST_VP(LOGICAL_CPU_NO)

Do (for CYCLE = 1 to NR_VP(LOGICAL_CPU_NO) and not end of READY_LIST(LOGICAL_CPU_NO)
If PROCESS = #RUNNING
    then
        RUNNING_LIST[VP_ID].PREEMPT := #FALSE
    else
        CANDIDATES := CANDIDATES + 1
    end if

VP_ID := VP_ID + 1
PROCESS := PROCESS.NEXT_PROCESS
end do

! Preempt appropriate candidates!
VP_ID := FIRST_VP(LOGICAL_CPU_NO)

Do for CHECK := 1 to NR_VP(LOGICAL_CPU_NO)
    If (RUNNING_LIST[VP_ID].PREEMPT = #TRUE) and (CANDIDATES > 0)
        then
            Call SET_PREEMPT(VP_ID)
            CANDIDATES := CANDIDATES - 1
        end if

VP_ID := VP_ID + 1
end do

LOGICAL_CPU_NO := LOGICAL_CPU_NO + 1
end do

Call UNLOCK(APT)

Return

End TC_ADVANCE

Figure 46: TC_ADVANCE Algorithm (Continued)
a process is found to be "running" then it should not be preempted as processes appear in the ready list in order of priority. When a running process is found, its associated entry in the preempt vector is marked "false." If a process is encountered in the "ready" state then it should be running and the "count" variable is incremented. When the first "n" processes have been checked or when we reach the end of the current ready list (whichever comes first), the entries in the preempt vector are "popped" from the stack. If an entry from the preempt vector is found to be "true", this indicates that its associated virtual processor is a candidate for preemption since it is either bound to a lower priority process, or it is "idle." In such a case, the "count" variable is evaluated to determine if the virtual processor associated with the vector entry should be preempted. If the count exceeds zero, a virtual preempt interrupt is sent to the VP and the count is decremented. Otherwise, no preempt is sent as there is no higher priority process awaiting scheduling.

This preemption algorithm is completed for every ready list in the Active Process Table. Once all ready lists have been evaluated, the APT is unlocked and control is returned to the caller. It is noted that it is not necessary to invoke TC_GETWORK before exiting ADVANCE. If the current VP requires rescheduling, it will have received a virtual preempt interrupt from the preemption algorithm. If this
has occurred, the VP will be rescheduled when its running process attempts to leave the Kernel domain and the virtual preempt interrupts are unmasked.

4. **Virtual_Preempt_Handler**

VIRTUAL_PREEMPT_HANDLER is the interrupt handler for virtual preempt interrupts. The entry address of VIRTUAL_PREEMPT_HANDLER is maintained in the virtual interrupt vector located in the Inner Traffic Controller. Once invoked, the handler locks the Active Process Table and determines which virtual processor is being preempted by calling RUNNING_VP. The process running on the preempted VP is then set to the "ready" state and TC_GETWORK is invoked to reschedule the virtual processor. When TC_GETWORK returns to VIRTUAL_PREEMPT_HANDLER, the APT is unlocked and a virtual interrupt return is executed. This return is simply a jump to the point in the hardware preempt handler where the virtual interrupts are unmasked. This effects a virtual interrupt return instruction.

5. **Remaining Procedures**

The remaining two procedures in the Traffic Controller module represent the extended instructions: PROCESS_CLASS and GET_DBR_NUMBER. Both procedures lock the Active Process Table and call RUNNING_VP to determine which virtual processor is executing the current process. The process ID (viz.,
APT entry Number) is then extracted from the running list. PROCESS_CLASS reads and returns the current process' security access classification from the APT. GET_DBR_NUMBER reads and returns the current process' DBR handle. It should be noted that in general the DBR number provided by procedure GET_DBR_NUMBER is only valid while the APT is locked. Particularly, in the current SASS implementation, the Segment Manager invokes GET_DBR_NUMBER and then passes the obtained DBR number to the Distributed Memory Manager for utilization at that level. In a more general situation, the process associated with the DBR number may have been unloaded before the DBR number was utilized, thus making it invalid. This problem does not arise in SASS as all processes remain loaded for the life of the system.

C. DISTRIBUTED MEMORY MANAGER MODULE

The Distributed Memory Manager module provides an interface between the Segment Manager and the Memory Manager process, manipulates event data in the Global Active Segment Table (G_AST), and dynamically allocates available memory. A detailed description of the Distributed Memory Manager interface to the Memory Manager process was presented by Wells [20]. The remaining extended instruction set is discussed in detail below. The complete PLZ/ASM source listings for the Distributed Memory Manager module is provided in Appendix C.
1. **MM_Read_Eventcount**

**MM_Read_Eventcount** is invoked by the Event Manager and the Traffic Controller to obtain the current value of the eventcount associated with a particular event. The input parameters to this procedure are a segment handle pointer and an instance (event Number), which together uniquely identify a particular event.

The G_AST is locked and the entry offset of the segment into the G_AST is obtained from the segment's handle. The instance parameter is then validated to determine which eventcount is to be read. If an invalid instance is specified, control is returned to the caller specifying an error condition. Otherwise, the current value of the specified eventcount is read. The G_AST is then unlocked, and the current eventcount value is returned to the caller.

2. **MM_Advance**

**MM_Advance** is invoked by the Traffic Controller to reflect the occurrence of some event. The input parameters to **MM_Advance** are a pointer to a segment's handle and a particular instance (event number).

The Global Active Segment Table is locked to prevent a race condition, and the offset of the segment's entry into the G_AST is obtained from the segment handle. The instance parameter is then validated to determine which eventcount is to be advanced. If an invalid instance is specified, an er-
ror condition is returned to the caller and no data entries are affected. If the instance value is valid, the appropriate eventcount is incremented, and its new value is returned.

3. **MM_Ticket**

**MM_Ticket** is invoked by the Event Manager to obtain the current value of the sequencer associated with a specified segment. The input parameter to **MM_Ticket** is a pointer to a segment's handle.

Initially, **MM_Ticket** locks the Global Active Segment Table to prevent a race condition. Next the offset of the segment's entry into the **G_AST** is obtained from the segment handle. The current value of the sequencer for the specified segment is then read and saved as a return parameter to the caller. The sequencer value is then incremented in anticipation of the next ticket request. Once this is complete, the **G_AST** is unlocked and control is returned to the caller.

4. **MM_Allocate**

The **MM_Allocate** procedure provided in this implementation is a stub of the **MM_Allocate** described in the Memory Manager design of Noble and Gary [5]. The primary function of **MM_Allocate** is the dynamic allocation of fixed size blocks of available memory space. It is invoked in the cur-
rent implementation by the initialization routines in
BOOTSTRAP_LOADER and TC_INIT for the allocation of memory
space used in the creation of the Kernel domain and Supervi-
sor domain stack segments and the creation of the Known Seg-
ment Tables for user processes. Dynamic reallocation of
previously used memory space (viz., garbage collection) is
not provided by the MM_ALLOCATE stub in this implementation.
All memory allocation required in this implementation is for
segments supporting system processes that remain active, and
thus allocated, for the entire life of the system. Memory
is allocated in blocks of 256 (decimal) bytes of processor
local memory (on-board RAM). In this stub allocatable memo-
ry is declared at compile time by a data structure
(MEM_POOL) that is accessible only by MM_ALLOCATE.

The input parameter to MM_ALLOCATE is the number of
blocks of requested memory. This parameter is converted
from a block size to the actual number of bytes requested.
This computation is made simple since memory is allocated in
powers of two. The byte size is obtained by logically
shifting left the input parameter eight times, where eight
is the power of two desired (viz., 256). Once the size of
the requested memory is computed, it is necessary to deter-
mine the starting address of the memory block(s) to be allo-
cated. To assist in this computation, a variable
(NEXT_BLOCK) is used to keep track of the next available
block of memory in MEM_POOL. NEXT_BLOCK, which is initial-
ized as zero, provides the offset into the memory being allocated. Once the starting address is obtained, the physical size of the memory allocated is added to NEXT_BLOCK so that the next request for memory allocation will begin at the next free byte of memory in MEM_POOL. This new value of NEXT_BLOCK is saved and the starting address of the memory for this request is returned to the caller.

D. GATE KEEPER MODULES

The SASS Gate Keeper provides the logical boundary between the Supervisor and the Kernel and isolates the Kernel from the system users, thus making it tamperproof. This is accomplished by means of the hardware system/normal mode and the software ring-crossing mechanism provided by the Gate Keeper. The Gate Keeper is comprised of two separate modules: 1) the USER_GATE module, and 2) the KERNEL_GATE_KEEPER module. These modules are disjoint, with the USER_GATE module residing in the Supervisor domain and the KERNEL_GATE_KEEPER module residing in the Kernel domain. It is important to note that the USER_GATE is a separately linked component in the Supervisor domain and is not linked to the Kernel. The only thing in common between these two modules is a set of constants identifying the valid extended instruction set which the Kernel provides to the users.

The Gate Keeper modules presented in this implementation are only stubs as they do not provide all of the functions
required of the Gate Keeper. However, the only task not provided in this implementation is the validation of parameters passed from the Supervisor to the Kernel. A detailed description of this parameter validation design is provided by Coleman [2]. In the process management demonstration, the Supervisor stubs are written in PLZ/ASM with all parameters passed by CPU registers. A detailed description of the Gate Keeper modules and the nature of their interfaces is presented below. The PLZ/ASM source listings for the two Gate Keeper modules are provided in Appendix D.

1. **User_Gate Module**

The USER_GATE module provides the interface structure between the user processes in the Supervisor domain and the Kernel. The USER_GATE is comprised of ten procedures (viz., entry points) that correlate on a one to one basis with the ten "user visible" extended instructions (listed in Figure 10) provided by the Kernel. The only action performed by each of these procedures is the execution of the "system call" instruction (SC) with a constant value, identifying the particular extended instruction invoked, as the source operand.

The SC instruction is a system trap that forces the hardware into the system mode (Kernel domain) and loads register 15 with the system stack pointer (Kernel domain stack). The current instruction counter value (IC) is
pushed onto the Kernel stack along with the current CPU flag control word (FCW). In addition, the system trap instruction is pushed onto the Kernel stack with the upper byte representing the SC instruction and the lower byte representing the SC instruction's source operand (viz., the Kernel extended instruction code). Together, these operations form an interrupt return (IRET) frame as illustrated in Figure 44. Once this is complete, the FCW is loaded with the FCW value found in the System Call frame of the Program Status Area (viz., the hardware "interrupt vector"). The structure of the Program Status Area is illustrated in Figure 47. The instruction counter is then loaded with the address of the SC instruction trap handler. This value is also located in the SC frame of the Program Status Area.
<table>
<thead>
<tr>
<th>OFFSET</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Unimplemented Instruction Trap</td>
</tr>
<tr>
<td>8</td>
<td>Privileged Instruction Trap</td>
</tr>
<tr>
<td>12</td>
<td>Kernel PCW</td>
</tr>
<tr>
<td>16</td>
<td>Segment Trap</td>
</tr>
<tr>
<td>20</td>
<td>Non-Maskable Interrupt</td>
</tr>
<tr>
<td>24</td>
<td>Kernel PCW</td>
</tr>
<tr>
<td>28</td>
<td>PHYS_PREEMPT_HANDLER</td>
</tr>
<tr>
<td>32</td>
<td>Vectored Int</td>
</tr>
</tbody>
</table>

*NOTE: Offsets represent Program Status Area structure for non-segmented 28002 microprocessor.*

Figure 47: Program Status Area
2. **Kernel_Gate_Keeper Module**

The system trap handler for the System Call instruction is the **KERNEL_GATE_KEEPER**. The address of the **KERNEL_GATE_KEEPER** and the Kernel FCW value are placed in the System Call frame of the Program Status Area by the **BOOTSTRAP_LOADER** module during initialization. The **KERNEL_GATE_KEEPER** fetches the extended instruction code from the trap instruction entry in the IRET frame on the Kernel stack. This value is then decoded by a "case" statement to determine which extended instruction is to be executed. If the extended instruction code is valid, the appropriate Kernel procedure is invoked. Otherwise, an error condition is set and no Kernel procedures are not invoked.

Once control returns to the **KERNEL_GATE_KEEPER**, the CPU registers and normal stack pointer (NSP) value are pushed onto the Kernel stack in preparation for return to the Supervisor domain. It is noted that this operation would normally occur immediately upon entry into the **KERNEL_GATE_KEEPER**. In this implementation, however, parameter validation is not accomplished and the CPU registers are used to pass parameters to and from the Kernel only for use by the process management demonstration. In an actual SASS environment, all parameters would be passed in a separate argument list and the CPU registers would appear exactly the same upon leaving the Kernel as they did upon entering the Kernel. This is
important to insure that no data or information is leaked from the Kernel by means of the CPU registers.

Control is returned to the Supervisor by means of the return mechanism in the hardware preempt handler. This mechanism is utilized to preclude the necessity of building a separate mechanism for the KERNEL_GATE_KEEPER that would actually perform the very same function. To accomplish this, the KERNEL_GATE_KEEPER executes an unconditional jump to the PREEMPT_RET label in PHYS_PREEMPT_HANDLER. This "jump" to the hardware preempt handler represents a "virtual IRET" instruction providing the same function as the virtual interrupt return described in the discussion of the virtual preempt handler. At this point, the virtual preempt interrupts are unmasked, the normal stack pointer and CPU registers are restored from the stack, and control is returned to the Supervisor by execution of the IRET instruction.

E. SUMMARY

The implementation of process management functions for the SASS has been presented in this chapter. The implementation was discussed in terms of the Event Manager, Traffic Controller, Distributed Memory Manager, and Gate Keeper modules.
Chapter XXIII

CONCLUSION

The implementation of process management for the security Kernel of a secure archival storage system has been presented. The process management functions presented provide a logical and efficient means of process creation, control, and scheduling. In addition, a simple but effective mechanism for inter-process communication, based on the eventcount and sequencer primitives, was created. Work was also completed in the area of Kernel database initialization and a Gate Keeper stub to allow for dual domain operation.

The design for this implementation was based on the Zilog Z8001 sixteen bit segmented microprocessor [22] used in conjunction with the Zilog Z8010 Memory Management Unit [23]. The actual implementation of process management for the SASS was conducted on the Advanced Micro Computers Am96/4116 MonoBoard Computer [1] featuring the AmZ8002 sixteen bit non-segmented microprocessor. Segmentation hardware was simulated by a software Memory Management Unit Image.

This implementation was effected specifically to support the Secure Archival Storage System (SASS) [17]. However, the implementation is based on a family of Operating Systems
designed with a primary goal of providing multilevel information security. The loop free modular design utilized in this implementation easily facilitates any required expansion or modification for other family members. In addition, this implementation fully supports a multiprocessor design. While the process management implementation appears to perform correctly, it has not been subjected to a formal test plan. Such a test plan should be developed and implemented before kernel verification is begun.

A. FOLLOW ON WORK

There are several possible areas in the SASS design that would be immediately suitable for continued research. In the area of hardware, this includes, the establishment of a multiprocessor environment, hardware initialization, and interfacing to the host computers and secondary storage. Further work in the Kernel includes the actual implementation of the memory manager process, and the refinement of the Gate Keeper and Kernel initialization structures. The implementation of the Supervisor has not been addressed to date. Its areas of research include the implementation of the File Manager and Input/Output processes, and the final design and implementation of the SASS-Hosts protocols.

Other areas that could also prove interesting in relation to the SASS include the implementation of dynamic memory management, the support of multilevel hosts, dynamic pro-
THE NAVAL POSTGRADUATE SCHOOL SECURE ARCHIVAL STORAGE SYSTEM. P-ETC(U)

M.A. Cox, R.R. Schell, S.L. Perdue

UNCLASSIFIED NPS52-81-001
cess creation and deletion, and the provision of constructive work to be performed by the Idle process.
Appendix A

EVENT MANAGER LISTINGS

Z8000ASM 2.02
LOC OBJ CODE SMT SRC STATEMENT

$LISTON $TTY

EVENT_MGR MODULE

CONSTANT
TRUE := 1
FALSE := 0
READ_ACCESS := 1
WRITE_ACCESS := 0
SUCCEEDED := 2
SEGMENT_NOT_KNOWN := 28
ACCESS_CLASS_NOT_EQ := 33
ACCESS_CLASS_NOT_GE := 41
KST_SEG_NO := 2
NR_OF_KSEGS := 10
MAX_NO_KST_ENTRIES := 54
NOT_KNOWN := $FF

TYPE
H_ARRAY ARRAY[3 WORD]

KST_REC RECORD
[MM_HANDLE H_ARRAY
  SIZE WORD
  ACCESS_MODE BYTE
  IN_CORE BYTE
  CLASS LONG
  N_SEG_NO SHORT INTEGER
  ENTRY_NUMBER SHORT INTEGER]

EXTERAL
MM_TICKET PROCEDURE
MM_READ_EVENTCOUNT PROCEDURE
TC_ADVANCE PROCEDURE
TC_AWAIT PROCEDURE
PROCSS_CLASS PROCEDURE
CLASS_EQ PROCEDURE
CLASS_GE PROCEDURE
ITC_GET_SEG_PTR PROCEDURE
$SECTION EN_KST_DCL

! NOTE: THIS SECTION IS AN "OVERLAY" OR "FRAME" USED TO DEFINE THE FORMAT OF THE KST. NO STORAGE IS ASSIGNED BUT RATHER THE KST IS STORED IN A SEPARATELY OBTAINED AREA. (A SEGMENT SET ASIDE FOR IT)!

$ABS 0

0000 KST ARRAY[MAX_NO_KST_ENTRIES KST_REC]
GLOBAL
$SECTION EM_GLB_PROC

READ PROCEDURE

* READS SPECIFIED EVENTCOUNT *
* AND RETURNS IT'S VALUE TO *
* THE CALLER *

* PARAMETERS: *
  * R1: SEGMENT # *
  * R2: INSTANCE *

* RETURNS: *
  * R0: SUCCESS CODE *
  * R4: EVENTCOUNT *

ENTRY

I SAVE INSTANCE I
PUSH $R15, R2

I "READ" ACCESS REQUIRED I
LD R2, #READ_ACCESS

I GET SEG HANDLE & VERIFY ACCESS I
CALL CONVERT_AND_VERIFY !RI:SEG #

R2:REQ. ACCESS
RETURNS:
R0:SUCCEESS CODE
R1:HANDLE PTR!

CP R0, #SUCCEEDED
IF EQ IACCESS PERMITTED!
THEN IREAD EVENTCOUNT!

IRESTORE INSTANCE!
POP R2, $R15
CALL MM_READ_EVENTCOUNT !R1:HPTR

R2:INSTANCE
RETURNS:
R0:SUCCEESS CODE
R4:EVENTCOUNT!

ELSE IRESTORE SP!
POP R2, $R15
FI

RET
END READ

- 249 -
0020 TICKET PROEDURE

* RETURNS CURRENT VALUE OF *
* TICKET TO CALLER AND INCRE- *
* MENTS SEQUENCER FOR NEXT *
* TICKET OPERATION *

* PARAMETERS: *
* R1: SEGMENT # *

* RETURNS: *
* R0: SUCCESS CODE *
* RR4: TICKET VALUE *

ENTRY

1 GET SEG HANDLE & VERIFY ACCESS !
1 "WRITE" ACCESS REQUIRED !

0020 2102 LD R2, #WRITE_ACCESS
0022 0000
0024 5F00 CALL CONVERT_AND_VERIFY R1:SEG #
0026 0000*

R2:ACCESS REQ
RETURNS:
R0:SUCCEESS CODE
R1:HANDEL PTR!

0028 0B00 CP R0, #SUCCEEDED
002A 0002

IF EQ 1 ACCESS PERMITTED1

002C 5E02 THEN 1 GET TICKET !
002E 0038*
0030 5F00 CALL MH_TICKET R1:HANDEL PTR
0032 0000*

RETURNS:
RR4:TICKET1

1 RSTORE SUCCEESS CODE I

0034 2100 LD R0, #SUCCEEDED
0036 0002

FI
0038 9E08 RET
003A END TICKET
AWAIT PROCEDURE

* CURRENT EVENTCOUNT VALUE IS *
* COMPARED TO USER SPECIFIED *
* VALUE. IF USER VALUE IS *
* GREATER THAN CURRENT EVENT- *
* COUNT VALUE THEN PROCESS IS *
"BLOCKED" UNTIL THE DESIRED *
* EVENT OCCURS. *

PARAMETERS:
* R1: SEGMENT #
* R2: INSTANCE (EVENT #)
* RR4: SPECIFIED VALUE

RETURNS:
* R0: SUCCESS CODE

ENTRY

I SAVE DESIRED EVENTCOUNT VALUE !
003A 91F4
PUSHL @R15, RR4
I SAVE INSTANCE I
003C 93F2
PUSH @R15, R2
I "READ" ACCESS REQUIRED !
003E 2102
LD R2, #READ_ACCESS
0040 0001
I GET SEG HANDLE & VERIFY ACCESS !
0042 5F00
CALL CONVERT_AND_VERIFY IR1:SEG #
R2:ACCESS REQ
RETURNS:
R0:SUCCESS CODE
R1:HANDLE PTR!

0046 0B00
CP R0, #SUCCEEDED
0048 0002
IF EQ I ACCESS PERMITTED I
004A 5E0E
THEN I AWAIT EVENT OCCURRENCE !
004C 005A
I RESTORE INSTANCE I
004E 97F2
POP R2, @R15
I RESTORE SPECIFIED VALUE !
0050 95F4
POPL RR4, @R15
0052 5F00
CALL TC_AWAIT IR1:HANDLE PTR
R2:INSTANCE
RR4:VALUE
RETURNS:
R0:SUCCESS CODE!
ELSE 1 RESTORE STACK1
POPL RR4, @R15
POP R2, @R15
RET
END AWAIT
ADVANCE PROCEDURE

* SIGNALS THE OCCURRENCE OF *
* SOME EVENT. EVENTCOUNT IS *
* INCREMENTED AND THE TRAFFIC *
* CONTROLLER IS INVOKED TO *
* AWAKEN ANY PROCESS AWAITING *
* THE OCCURRENCE. *

PARAMETERS:
* R1: SEGMENT # *
* R2: INSTANCE (EVENT #) *

RETURNS:
* RO: SUCCESS CODE *

PARAMETERS:
* R1: SEGMENT # *
* R2: INSTANCE (EVENT #) *

RETURNS:
* RO: SUCCESS CODE *

ENTRY

SAVE INSTANCE !
PUSH @R15, R2

GET SEG HANDLE & VERIFY ACCESS !
"WRITE" ACCESS REQUIRED !
LD R2, WRITE_ACCESS
CALL CONVERT_AND_VERIFY !R1:SEG #
R2:ACCESS REQ
RETURNS:
RO:SUCCESS CODE
R1:HANDLE PTR!

IF EQ ! ACCESS PERMITTED !
THEN ! ADVANCED EVENTCOUNT !

RESTORE INSTANCE !
POP R2, @R15
CALL TC_ADVANCE !R1:HANDLE PTR
R2:INSTANCE
RETURNS:
RO:SUCCESS CODE!

ELSE !RESTORE STACK!

RET
END ADVANCE
CONVERT_AND_VERIFY

**PROCEDURE**

* CONVERTS SEGMENT NUMBER TO KST INDEX*
* AND EXTRACTS SEGMENT'S HANDLE FROM *
* KST. IF SUCCESSFUL, THEN ACCESS *
* CLASS OF SUBJECT IS CHECKED AGAINST *
* ACCESS CLASS OF OBJECT TO INSURE *
* THAT ACCESS IS PERMITTED.

**PARAMETERS:**
* R1: SEGMENT NUMBER
* R2: ACCESS REQUESTED

**RETURNS:**
* R0: SUCCESS CODE
* R1: HANDLE POINTER

**ENTRY**

I SAVE REQUESTED ACCESS!

`0000 93F2` PUSH R15, R2

I GET SEGMENT HANDLE!

`0002 5F00` CALL GET_HANDLE 1R1:SEG #

**RETURNS:**
R0: SUCCESS CODE
R4: HANDLE PTR
R5: CLASS PTR!

`0006 0B00` CP R0, #SUCCEEDED
`0008 0002` IF EQ ! SEGMENT IS KNOWN!

`000A 5E02` THEN ! VERIFY ACCESS!

`000C 005E` ! SAVE HANDLE & CLASS PTR!

`000E 91F4` PUSHL R0, RR4

I GET SUBJECT'S SAC!

`0010 5F00` CALL PROCESS_CLASS !RETURNS:

RR2:PROC CLASS!

`0012 0000` ! RETRIEVE SEG CLASS POINTER!

`0014 95F0` POPL RR0, R0R15

I GET SEGMENT'S CLASS!

`0016 1414` LDL RR4, R1

I RETRIEVE REQUESTED ACCESS!

`0018 97F1` POP R1, R15

I SAVE HANDLE POINTER!

`001A 93F0` PUSH R15, R0

I CHECK ACCESS CLEARANCE!

`001C 0B31` CP R1, #WRITE_ACCESS

`001E 0000` IF EQ ! WRITE ACCESS REQUESTED!
THEN
CALL CLASS_EQ !BR2:PROCESS CLASS

BR4:SEGMENT CLASS
RETURNS:
R1: CONDITION CODE!

CP R1, #FALSE
IF EQ !ACCESS NOT PERMITTED!
THEN
LD R0, #ACCESS_CLASS_NOT_EQ
ELSE !ACCESS PERMITTED!
LD R0, #SUCCEEDED

FI
ELSE !READ ACCESS REQUESTED !
CALL CLASS_GE !BR2:PROCESS CLASS

BR4:SEGMENT CLASS
RETURNS:
R1: CONDITION CODE!

CP R1, #FALSE
IF EQ !ACCESS NOT PERMITTED!
THEN
LD R0, #ACCESS_CLASS_NOT_GE
ELSE !ACCESS PERMITTED!
LD R0, #SUCCEEDED

FI
! RETRIEVE HANDLE POINTER !
POP R1, R15
ELSE
! RESTORE STACK !
POP R2, R15
FI
RET
END_CONVERT_AND_VERIFY
GET_HANDLE
PROCEDURE

*** CONVERTS SEGMENT NUMBER TO ***
* KST INDEX AND DETERMINES IF *
* SEGMENT IS KNOWN. IF KNOWN *
* POINTER TO SEGMENT HANDLE *
* AND POINTER TO SEGMENT CLASS* *
* ARE RETURNED. *

**********************************************************************
* PARAMETERS: *
* R1: SEGMENT NUMBER *
**********************************************************************
* RETURNS: *
* RO: SUCCESS CODE *
* R4: HANDLE POINTER *
* R5: CLASS POINTER *
***********************************************************************

ENTRY
CONVERT SEGMENT # TO KST INDEX # 1

0062 0301 SUB R1, #NR_OF_KSEGS
0064 000A
 : VERIFY KST INDEX :
0066 2100 LD R0, #SUCCEEDED
0068 0002 CP R1, #0
006A 0801 IF LE INDEX NEGATIVE! THEN
006C 5E0A 006D 007A 007E 5E08
0072 2100 LD R0, #SEGMENT_NOT_KNOWN
0074 001C ELSE INDEX POSITIVE! 
0076 5E08 0078 0086 CP R1, #MAX_NO_KST_ENTRIES
007A 0801 IF GT INDEX EXCEEDS MAXIMUM INDEX! THEN INVALID INDEX!
007C 0036 007E 5E02 0080 0086
0082 2100 LD R0, #SEGMENT_NOT_KNOWN
0084 001C FI
0086 0800 CP R0, #SUCCEEDED
0088 0002 IF EQ INDEX VALID! THEN
008A 5E0E 008C 00BE
008C 00BE IF SAVE KST INDEX !
008E 93F1 PUSH &R15, R1
0090 2101 IF GET KST ADDRESS! 
0092 0002 LD R1, #KST_SEG_NO
0094 5F00 CALL ITC_GET_SEG_PTR [R1:KST_SEG_NO
- 256 -
RETURNS:
RO: KST ADDR

I RETRIEVE KST INDEX #
098 97F3
POP R3, @R15

I CONVERT KST INDEX # TO KST OFFSET!
09A 1902
MULT RB2, #SIZEOF KST_REC

I COMPUTE KST ENTRY ADDRESS!
09C 0010
ADD R3, RO

I SEE IF SEGMENT IS KNOWN!
09E 8103
CP KST.M_SEG_NO(R3), #NOT_KNOWN

IF EQ ISEGMENT NOT KNOWN!
THEN
0A0 4D31
CP KST.NSEG_NO(R3), #NOTKNOWN

LOOK2 000E

IF EQ ISEGMENT NOT KNOWN!
THEN
0A4 00FF

ELSE ISEGMENT KNOWN!
0A6 5E02
0A8 00B2'
0AA 2100
0AC 001C
0AE 5E08
0B0 00BE'
0B2 2100
0B4 0002

I GET HANDLE POINTER!
0B6 7634
LDA R4, KST.MM_HANDLE(R3)
0B8 0000

I GET CLASS POINTER!
0BA 7635
LDA R5, KST.CLASS(R3)
0BC 000A

FI
FI

0BE 9E08
RE'T
0CC0
END GET_HANDLE
END EVENT_MGR

- 257 -
Appendix B

TRAFFIC CONTROLLER LISTINGS

Z8000ASM 2.02
LOC OBJ CODE STMT SOURCE STATEMENT

$LISTON STTY
TC MODULE

CONSTANT

! ******** SYSTEM PARAMETERS ********
NR_PROC := 4
VP_NR := 2
NR_CPU := 2
NR_KST := 54

! ******** SYSTEM CONSTANTS ********
RUNNING := 0
READY := 1
BLOCKED := 2
IDLE_PROC := %DDDD
NIL := %FFFF
INVALID := %EEEE
KERNEL_STACK := 1
USER_STACK := 3
KST_SEG := 2
KST_LIMIT := 1
USER_FCW := %1800
WRITE := 0

! INDICATES LOWEST SYSTEM SECURITY CLASS!
SYSTEM_LOW := 0
STK_OFFSET := %FF
REMOVED := %ABCD
TRUE := 1
FALSE := 0
SUCCEEDED := 2

TYPE
AP_PTR WORD
VP_PTR WORD
ADDRESS WORD
B_ARRAY ARRAY[3 WORD]
AP_TABLE RECORD
[ NEXT_AP AP_PTR
  DBR WORD
  SAC LONG
  PRI INTEGER
  STATE INTEGER
  AFFINITY WORD
  VP_ID VP_PTR
  HANDLE H_ARRAY
  INSTANCE WORD
  VALUE LONG
  FILL_2 ARRAY[2 WORD]
]
RUN_ARRAY ARRAY[VPNR AP_PTR]
RDY_ARRAY ARRAY[NRCPU AP_PTR]
AP_DATA ARRAY[NR_PROC AP_TABLE]
VP_DATA RECORD
[ NR_VP ARRAY[NR_CPU WORD]
  FIRST ARRAY[NR_CPU VP_PTR]
]
KST_REC RECORD
[ H_ARRAY H_ARRAY
  SIZE WORD
  ACCESS BYTE
  IN_CORE BYTE
  CLASS LONG
  M_SEG_NO SHORT_INTEGER
  ENTRY_NUM SHORT_INTEGER
]
EXTERNAL
K_LOCK PROCEDURE
K_UNLOCK PROCEDURE
SET_PREEMPT PROCEDURE
SWAP_VDBR PROCEDURE
IDLE PROCEDURE
RUNNING_VP PROCEDURE
CREATE_INT_VEC PROCEDURE
LIST_INSERT PROCEDURE
ALLOCATE_MMU PROCEDURE
MM_ALLOCATE PROCEDURE
UPDATE_MMU_IMAGE PROCEDURE
CREATE_STACK PROCEDURE
MM_ADVANCE PROCEDURE
MM_READ_EVENTCOUNT PROCEDURE
G_AST_LOCK WORD
PREEMPT_RET LABEL
$SECTION TC_DATA
INTERNAL
0000 APT RECORD
    [ LOCK WORD
      RUNNING_LIST RUN_ARRAY
      READY_LIST RDY_ARRAY
      BLOCKED_LIST AP_PTR
      FILL_3 LONG
      VP VP_DATA
      FILL ARRAY[4 WORD]
      AP AP_DATA
    ]

!THESE VARIABLES ARE USED DURING TC
INITIALIZATION TO SPECIFY AVAILABLE
ENTRIES IN THE APT, AND ARE INITIAL-
IZED BY TC_INIT IN THIS IMPLEMENTATION!

00A0 NEXT_VP WORD
00A2 APT_ENTRY WORD

$SECTION TC_LOCAL
$ABS 0
!NOTE: USED AS OVERLAY ONLY!
0000 ARG_LIST RECORD
    [ REG ARRAY[13 WORD]
      IC WORD
      CPU_ID WORD
      SAC1 LONG
      PRI1 WORD
      USR_STK WORD
      KER_STK WORD
      KST1 LONG
    ]

$ABS 0
!NOTE: USED AS STACK FRAME FOR
STORAGE OF TEMPORARY VARIABLES
FOR CREATE_PROCESS!

0000 CREATE RECORD
    [ ARG_PTR WORD
      DBR_NUM WORD
      LIMITS WORD
      SEG_ADDR ADDRESS
      N_S_P WORD
    ]

$ABS 0

0000 HANDLE_VAL RECORD
    [ HIGH LONG
      LOW WORD
    ]

!THE FOLLOWING DECLARATION IS UTILIZED
AS A STACK FRAME FOR STORAGE OF

- 260 -
TEMPORARY VARIABLES UTILIZED BY 
TC_ADVANCE AND TC_AWAIT.!

$ABS 0

0000 TEMP RECORD

HANDLE_PTR WORD
EVENT_NR WORD
EVENT_VAL LONG
ID_Flag WORD
CPU_NUM WORD
HANDLE_HIGH LONG
HANDLE_LOW WORD

$SECTION TC_KST_DCL

!NOTE: KST DECLARATION IS USED HERE 
TO SUPPORT KST INITIALIZATION FOR 
THIS DEMONSTRATION ONLY. THIS 
DECLARATION AND INITIALIZATION 
SHOULD EXIST AT THE SEGMENT MANAGER 
LEVEL AND THUS SHOULD BE REMOVED 
UPON IMPLEMENTATION OF SYSTEM 
INITIALIZATION.!

$ABS 0

0000 KST ARRAY[NR_KST KST_REC]
$SECTION TC_INT_PROC

TC_GETWORK PROCEDURE

* PROVIDES GENERAL MANAGEMENT OF USER PROCESSES BY EFFECTING PROCESS SCHEDULING ON VIRTUAL PROCESSORS

* PARAMETERS:
  * R1: CURRENT VP ID
  * R3: LOGICAL CPU #

* LOCAL VARIABLES:
  * R2: NEXT READY PROCESS
  * R4: AP PTR

ENTRY
  IF FIND FIRST READY PROCESS

0000 6132    LD R2, APT.READY_LIST(R3)
0002 0006*    GET_READY_AP:

0004 3B02    CP R2, #NIL
0006 FFFF
0008 5E0E    IF EQ !NO READY PROCESS! THEN
000A 0010*
000C 5E08    EXIT FROM GET_READY_AP
000E 0026*

0010 4D21    CP APT.AP.STATE(R2), #READY
0012 002A*
0014 0001
0016 5E0E    IF EQ !PROCESS READY! THEN
0018 001E*
001A 5E08    EXIT FROM GET_READY_AP
001C 0026*

001E 6124    ! GET NEXT AP FROM LIST!
0020 0020*
0022 A142    LD R4, APT.AP.NEXT_AP(R2)
0024 B8EF    OD
0026 0B02    CP R2,#NIL
0028 FFFF
002A 5E0E    IF EQ !IF NO PROCESSES READY! THEN
002C 003C*

002E 4D15    ! LOAD IDLE PROCESS!
0030 0002*
0032 DDDD
0034 5F00    CALL IDLE
0036 0000*    ELSE
0038 5E08
I LOAD FIRST READY AP I

LD APT.RUNNING_LIST(R1), R2

LD APT.AP.STATE(R2), #RUNNING

LD APT.AP.VP_ID(R2), R1

LD R1, APT.AP.DBR(R2)

CALL SWAP_VDBR !(R1:DBR)

FI

RET

END TC_GETWORK
VIRTUAL_PREEAPT_HANDLER PROCEDURE

* LOADS FIRST READY AP *
* IN RESPONSE TO PREAMPT *
* INTERRUPT *
*************************************************

ENTRY

!! CALL WAIT_LOCK (APT-.LOCK) **!
!! RETURNS WHEN PROCESS HAS LOCKED APT **!
ENTRY

LDA R4, APT.LOCK
CALL K_LOCK

! GET RUNNING_VP ID !
CALL RUNNING_VP !RETURNS:

R1:VP_ID
R3:CPU #1

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

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

! LOAD FIRST READY PROCESS !
CALL TC_GETWORK !R1:VP_ID
R3:CPU #1

!NOTE: THIS IS THE INITIAL POINT OF
EXECUTION FOR USER PROCESSES.!

VIRT_PREAMPT_RETURN:
!! CALL UNLOCK (APT-.LOCK) **!
!! RETURNS WHEN PROCESS HAS UNLOCKED APT **!
!! AND ADVANCED ON THIS EVENT **!
ENTRY

LDA R4, APT.LOCK
CALL K_UNLOCK

! PERFORM A VIRTUAL INTERRUPT RETURN !
!NOTE: THIS JUMP EFFECTS A VIRTUAL
IRET INSTRUCTION.!
JP PREAMPT_RET
GLOBAL
$SECTION TC_GLB_PROC

0000    TC_INIT    PROCEDURE

*                     * INITIALIZES APT HEADER *
*                     * AND VIRTUAL INT VECTOR *

* PARAMETERS:         *
* R1: CPU_ID          *
* R2: NR_VP           *

ENTRY

! NOTE: THE NEXT FOUR VALUES ARE
! ONLY TO BE INITIALIZED ONCE. !
LD     #IT_VP, #0

0002    00A0*
0004    0000*
0006    4D05
0008    00A2*
00A0    0000*
00C0    4D05
00D0    00A1*
0100    FFFF
0112    4D08
0114    0000*

CLR APT.LOCK

! ***********************
! NOTE: THE FOLLOWING CODE IS INCLUDED
! ONLY FOR SIMULATION OF A MULTIPROCESSOR
! ENVIRONMENT. THIS IS TO INSURE THAT THE
! READY LIST(S) AND VP DATA OF THE SIMULATED
! CPU(S) ARE PROPERLY INITIALIZED. IN AN
! ACTUAL MULTIPROCESSOR ENVIRONMENT, THIS
! BLOCK OF CODE SHOULD BE REMOVED.

0116    2104
0118    0000*

LD     R4, #0

DO

011A    0804
011C    0004

CP     R4, #NR_CPU*2

011E    5E02
0220    0026*
0222    5E08
0224    0036*

IF EQ !ALL LISTS INITIALIZED!
THEN EXIT

FI

! INITIALIZE READY_LISTS AS EMPTY !
LD     APT.READY_LIST(R4), #NIL

0226    4D45
0228    0006*
022A    FFFF

! INITIALLY MARK ALL LOGICAL CPU'S
! AS HAVING 1 VP. THIS IS NECESSARY
! TO INSURE TC ADVANCE WILL FUNCTION
! PROPERLY, AS IT EXPECTS EVERY CPU
TO HAVE AT LEAST 1 VP.

LD APT.VP.NR_vp(R4), #1

INC R4, #2

OD

! END MULTIPROCESSOR SIMULATION CODE.

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

LD APT.VP.NR_vp(R1), R2

LD R3, NEXT_vp

LD APT.VP.FIRST(R1), R3

LD APT.VP.FIRST(R1), R3

LD R3, APT.VP.FIRST(R1)

LD R5, R2

MULT RR4, #2

ADD R3, R5

LD NEXT_vp, R3

LD R5, R2

LD APT.RUNNING_LIST(R3), #IDLE_PROC

INC R3, #2

DEC R2, #1

OD

LD APT.READY_LIST(R1), #NIL

LD R1, #0

ENTRY ADDRESS!

LDA R2, VIRTUAL_PREEMPT_HANDLER

CALL CREATE_INT_VEC

!R1:VIRTUAL INTERRUPT #
R2:INTERUPT HANDLER ADDRESS!

RET

END TC_INIT

- 267 -
CREATE_PROCESS PROCEDURE

* CREATES USER PROCESS *
* DATABASES AND APT *
* ENTRIES *

* PARAMETERS: *
* R14: ARGUMENT PTR *

ENTRY

NOTE: THIS PROCEDURE IS A STUB TO ALLOW
PROCESS INITIALIZATION FOR THIS
DEMONSTRATION.

ESTABLISH STACK FRAME FOR LOCAL
VARIABLES.

SUB R15, #SIZEOF CREATE

I STORE INPUT ARGUMENT POINTER I
LD CREATE.ARG_PTR(R15), R14

I LOCK APT I
LDA R4, APT.LOCK
CALL K_LOCK

I RETURNS WHEN APT IS LOCKED I
I CREATE MMU ENTRY FOR PROCESS I
CALL ALLOCATE_MMU !RETURNS:

R0: DBR #1
I GET NEXT AVAILABLE ENTRY IN APT I
LD R1, APT_ENTRY

I COMPUTE APT OFFSET I
LD R2, #SIZEOF AP_TABLE
ADD R2, R1
I SAVE NEXT AVAILABLE APT ENTRY I
LD APT_ENTRY, R2

I CREATE APT ENTRY FOR PROCESS I
LD APT.AP.NEXT_AP(R1), #NIL

LD APT.AP.DBR(R1), R0

I GET PROCESS CLASS I
LDL RR2, ARG_LIST.SAC1(R14)

I GET PROCESS PRIORITY I
LD R2, ARG_LIST.PRI1(R14)
```
00B4 0022  LD  APT.AP_PRI(R1), R2
00B6 6F12  ! GET LOGICAL CPU # !
00B8 0028'  I SET LOGICAL CPU #!
00BA 61E2  LD  R2, ARG_LIST.CPU_ID(R14)
00BC 001C
00BE 6F12  LD  APT.AP.AFFINITY(R1), R2
00C0 002C'  I THREAD IN LIST AND MAKE READY!
00C2 7623  LD  R3, APT.READY_LIST(R2)
00C4 0006'
00C6 7604  LDA  R4, APT.AP.NEXT_AP
00C8 0023'
00CA 7605  LDA  R5, APT.AP.PRI
00CC 0028'
00CE 7606  LDA  R6, APT.AP.STATE
00D0 002A'
00D2 2107  LD  R7, #READY
00D4 0001
00D6 AD21  EX  R1, R2
00D8 6F00  LD  CREATE.DBJUM(R15), R0
00DA 0002
00DC 5F00  CALL  LIST_INSERT
00DE 0000*  1R2: OBJ ID
00E0 7604  LDA  R3, LIST HEAD PTR
00E2 0000'
00E4 5F00  CALL  K_UNLOCK
00E6 0000*  1CREATE USER STACK!
00E8 61F2  LD  R14, CREATE.ARG_PTR(R15)
00EA 0000
00E2 61E3  LD  R3, ARG_LIST.USR_STK(R14)
00EE 0024
00F0 6F03  ! SAVE LIMITS !
00F2 0004  LD  CREATE.LIMITS(R15), R3
00F4 5F00  CALL  MM_ALLOCATE(R3): # OF BLOCKS
00FA 0000*  RETURNS:
00F8 A128  R2: START ADDR!
00F8 0004  I COMPUTE & SAVE NSPI!
00F2 0004  LD  R8, R2
00F4 5F00  I ESTABLISH INITIAL SP VALUE
00F6 0000  FOR USER STACK. !
00FA 00108  ADD  R8, #STK_OFFSET
```
00FC 00FF  LD  CREATE_M_S_P(R15), R8
00FF 6FF8  ! RESTORE LIMITS !
0100 0008  LD  R4, CREATE_LIMITS(R15)
0102 61F4  ! RESTORE DBR !
0104 0004  LD  R0, CREATE_DBR_NUM(R15)
0106 AB40  DEC  R4, ISEG_LIMITS!
0108 61F0  IRESTORE DBR!
010A 0002  LD  R1, #USER_STACK
010C 2101  LD  R3, #WRITE_ATTRIBUTE!
0110 2103  CALL UPDATE_MNU_IMAGE
0114 5F00  ICREATE KERNEL STACK!
0116 0000*  I RESTORE ARGUMENT POINTER !

0118 61FE  LD  R14, CREATE_ARG_PTR(R15)
011A 0000  LD  R3, ARG_LIST_KER_STK(R14)
011C 61E3  LD  R1, #KERNEL_STACK
011E 0026  LD  R4, R3
0120 5F00  CALL MM_ALLOCATE !R3: # OF BLOCKS
0122 0000*  RETURNS
           R2: START ADDR!

0124 61F0  LD  R0, CREATE_DBR_NUM(R15)
0126 0002  LD  R1, #KERNEL_STACK
0128 2101  LD  R4, R3
012A 0001  DEC  R4
012C AB40  LD  R3, #WRITE
0130 2103  I SAVE START ADDRESS !
0134 6FF2  LD  CREATE_SEG_ADDR(R15), R2
0136 0006  CALL UPDATE_MNU_IMAGE
0138 5F00  ICREATE KERNEL STACK!
013A 0000*  I RESTORE ARGUMENT POINTER !
I RESTORE STACK ADDRESS

LD R1, CREATE_SEG_ADDR(R15)

0140 61F1

0142 0006

0144 2103

0146 1800

0148 61E4

014A 001A

I RESTORE INITIAL NSP

LD R5, CREATE_N_SP(R15)

014C 61F5

014E 0008

0150 7606

0152 0076

0154 030F

0156 0008

0158 1CF9

015A 0303

I LOAD ARGUMENT POINTER FOR CREATE_STACK CALL

LD R0, R15

015C A1F0

PUSH @R15, R1

015E 93F1

LD R1, R14

I LOAD INITIAL REGISTER VALUES TO BE PASSED TO USER PROCESS AS INITIAL PARAMETERS

LD R2, ARG_LIST.IN(R1), #13

0162 5C11

0164 020C

0166 0000

0168 97F1

CALL CREATE_STACK

016A 5F00

016C 0000

I R0: ARGUMENT PTR
R1: TOP OF STACK
R2-R14: INITIAL REG STATES

I NOTE: THE ABOVE INITIAL REG STATES REPRESENT THE INITIAL PARAMETERS (VIZ., REGISTER CONTENTS) THAT A USER PROCESS WILL RECEIVE UPON INITIAL EXECUTION

016E 010F

ADD R15, #8 1OVERRIDE PARAMETERS

0170 0008

I ALLOCATE KST

LD R3, #KST_LIMIT

0172 2103

0174 0001

0176 5F00

CALL MM_ALLOCATE (R3): # OF BLOCKS

0178 0000

REURNS
R2: START ADDR

017A 61F0

LD R0, CREATE_DBR_NUM(R15)

017C 0002

I SAVE KST ADDRESS

017E 6FF2

LD CREATE_SEG_ADDR(R15), R2
0180 0006  "MAKE MHU ENTRY FOR KST SEG!
0182 2101  LD    R1, #KST_SEG
0184 0002
0186 2103  LD    R3, #WRITE_ATTRIB
0188 0000
018A 2104  LD    R4, #KST_LIMIT-1
018C 0000
018E 5F00  CALL  UPDATE_MHU_IMAGE
0190 0000*

  !R0: DBR #
  R1: SEGMENT #
  R2: SEG ADDRESS
  R3: SEG ATTRIBUTES
  R4: SEG LIMITS!

  !RESTORE KST ADDRESS!

0192 61F2  LD    R2, CREATE_SEG_ADDR(R15)
0194 0006
0196 5F00  CALL  CREATE_KST R2:KST ADDR!
0198 01A0'

  !REMOVE TEMPORARY VARIABLE
  STACK FRAME!

019A 010F  ADD   R15, #$SIZEOF CREATE
019C 000A
019E 9E08  RET
01A0    END CREATE_PROCESS

- 272 -
CREATE_KST  PROCEDURE

* CREATES KST STUB FOR *
* PROCESS MANAGEMENT *
* DEMO. INSERTS ROOT *
* ENTRY IN KST. NOT *
* INTENDED TO BE FINAL *
* PRODUCT. *

PARAMETERS:
* R2: KST ADDRESS *

ENTRY

NOTE: THIS PROCEDURE IS A STUB USED FOR INITIALIZATION IN THIS IMPLEMENTATION ONLY. THE ACTUAL INITIALIZATION CODE FOR THE KST WILL RESIDE AT THE SEGMENT MANAGER LEVEL ONCE IMPLEMENTATION OF SYSTEM INITIALIZATION IS EFFECTED.

CREATE ROOT ENTRY IN KST!

LDL RR6, #-1 !ROOT HANDLE!

LDL KST.MM_HANDLE(R2), RR6

ISET ROOT ENTRY # IN G_AST !
LD KST.MM_HANDLE[2](R2), #0

I SET ROOT CLASSIFICATION !
LDL RR6, #SYSTEM_LOW

LDL KST.CLASS(R2), RR6

ISET MENTOR SEG #!
LDB KST.M_SEG_NO(R2), #0

I INITIALIZE FREE KST ENTRIES FOR DEMO. NOT FULL KST!

LD R1, #10

DO
CP R1, #0

IF EQ THEN EXIT FI

ADD R2, #SIZEOF KST_REC

- 273 -
01D4 4C25       LDB  KST.M_SEG_NO(R2), #FF
01D6 000E       
01D8 FFFF       
01DA AB10       DEC  R1
01DC EF36       OD
01DE 9E08       RET
01E0            END CREATE_KST

- 274 -
TC_Advance

PROCEDURE

* EVENTCOUNT IS ADVANCED BY *
* INVOCATION OF MM_ADVANCE. *
* PROCESSES THAT ARE AWAITING *
* THIS EVENT OCCURRENCE ARE *
* REMOVED FROM THE BLOCKED LIST* *
* AND MADE READY. THE READY *
* LISTS ARE THEN CHECKED TO *
* INSURE PROPER SCHEDULING IS *
* EFFECTED. IF NECESSARY VIR-
* TUAL PREEMPTS ARE SENT TO ALL*
* THOSE VP'S BOUND TO LOWER *
* PRIORITY PROCESSES. *

* PARAMETERS: *
* R1: HANDLE POINTER *
* R2: INSTANCE (EVENT #) *

* RETURNS: *
* R0: SUCCESS CODE *

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

ENTRY

! ESTABLISH TEMPORARY VARIABLE
STACK FRAME. !

01E0 030F  SUB R15, #SIZEOF TEMP
01E2 0012  ! SAVE INPUT ARGUMENTS !
01E4 6FF1  LD TEMP.HANDLE_PTR(R15), R1
01E6 0000 01E8 6FF2  LD TEMP.EVENT_NR(R15), R2
01EA 0002  ! LOCK APT !
01EC 7604  LDA R4, APT.LOCK
01EE 0000* 01F0 5F00  CALL K_LOCK
01F2 0000* 01F4 5F00  ! RETURNS WHEN APT IS LOCKED !
01F6 0000*  ! ANNOUNCE EVENT OCCURRENCE BY *
01F8 0800  ! INCREMENTING EVENTCOUNT IN G_AST!
01FA 0002  CALL MM_ADVANCE IR1:HANDLE_PTR
01FC 5E0E  R2:INSTANCE
01FE 0372'  RETURNS:
0200 0DF2  R0:SUCCESS CODE
0202 0004  RR2:EVENTCOUNT!

- 275 -
I RESTORE INSTANCE !
0204 61F0   LD  R0, TEMP.EVENT_NR(R15)
0206 0002
I RESTORE HANDLE POINTER !
0208 61F1   LD  R1, TEMP.HANDLE_PTR(R15)
020A 0000
I SAVE HANDLE !
020C 5414   LDL  RR4, HANDLE_VAL.HIGH(R1)
020E 0000
0210 5D74   LDL  TEMP.HANDLE_HIGH(R15), RR4
0212 000C
0214 6114   LD  R4, HANDLE_VAL.LOW(R1)
0216 0004
0218 61F4   LD  TEMP.HANDLE_LOW(R15), R4
021A 0010
I AWAKE ALL PROCESSES AWAITING THIS EVENT OCCURRENCE !
021C 6101   LD  R1, APT.BLOCKED_LIST
021E 000A.  WAKE_UP:
0220 7606   LDA  R6, APT.BLOCKED_LIST
0222 000A.  DO
0224 0B01   ! DETERMINE IF AT END OF BLOCKED LIST !
0226 PFFF   CP   R1, #NIL
0228 5E0E   ! IF EQ ! NO MORE BLOCKED PROCESSES !
022A 0230.  THEN EXIT FROM WAKE_UP
022C 5E08
022E 02B4.  FI
0230 6117   ! SAVE NEXT ITEM IN LIST !
0232 0020.  LD   R7, APT.AP.NEXT_AP(R1)
0234 5414   ! DETERMINE IF PROCESS IS ASSOCIATED WITH CURRENT HANDLE !
0236 000C   LDL  RR4, TEMP.HANDLE_HIGH(R15)
0238 5014   CPL  RR4, APT.AP.HANDLE(R1)
023A 0030.  IF EQ ! HIGH HANDLE VALUE MATCHES !
023C 5E0E   THEN
023E 02A2.  0240 61F4   LD  R4, TEMP.HANDLE_LOW(R15)
0242 0010
0244 4B14   CP   R4, APT.AP.HANDLE[2](R1)
0246 0034.  IF EQ ! HANDLE'S MATCH !
0248 5E0E   THEN ! CHECK FOR INSTANCE MATCH !
024A 029C.  024C 61F0   LD   R0, TEMP.EVENT_NR(R15)
024E 0002
- 276 -
0250 4810  CP   R0, APT.AP.INSTANCE(R1)
0252 0036
0254 5E0E  IF EQ ! INSTANCE MATCHES !
0256 0296
0258 54F2  THEN ! DETERMINE IF THIS IS THE
0260 0036
0262 0290
0264 2F67  OCCURRENCE THE PROCESS
0266 91F6  WAITING FOR !
0268 6112  LDL   RR2, TEMP.EVENT_VAL(R15)
026A 002C
026C 7623  CPL   RR2, APT.AP.VALUE(R1)
026E 0006
0270 7604
0272 0020
0274 7605
0276 0028
0278 7606
027A 002A
027C 2107
027E 0001
0280 A112  LD     R2, R1
0282 5F00  CALL   LIST_INSERT
0284 0000*
0286 95F6  ! R2: OBJ ID
0288 2108  ! R3: LIST HEAD PTR
028A ABCD  ! R4: NEXT OBJ PTR
028C 5E08  ! R5: PRIORITY PTR
028E 0292  ! R6: STATE PTR
0290 8DB8  ! R7: STATE VALUE !
0292 5E08
0294 0298*
0296 8DB8  ! RESTORE LOCAL VARIABLES !
0298 5E08
029A 029E*

0292 5E08  ELSE ! PROCESS STILL BLOCKED!
0294 0298*
0296 8DB8  CLR   R11
0298 5E08  FI ! END INSTANCE CHECK !
029A 029E*

- 277 -
CLR  R11
FI  ! END HANDLE CHECK !
ELSE  ! PROCESS STILL BLOCKED!
CLR  R11
FI  ! END HIGH HANDLE CHECK !
! RESET AP POINTER REGISTERS !
CP   R11,   #REMOVED
IF  ! PROCESS IS STILL BLOCKED !
THEN
LDA   R6, AP.I.AP.NEXT_AP(R1)
FI
LD   R1, R7
OD
! DETERMINE IF ANY VIRTUAL PREEMPT INTERRUPTS ARE REQUIRED !
CLR  R2
PREEMPT_CHECK:
DO
CP   R2,   #NR_CPU * 2
IF  ! ALL READY LISTS CHECKED ! THEN
EXIT FROM PREEMPT_CHECK
FI
CLR  R1
DO  FOR  R1=1 TO NR_VP'S!
INC  R1
CP   R1, AP.VP.NR_VP(R2)
IF  ! PREEMPT VECTOR COMPLETED ! THEN EXIT
CLR  R3
LD   R4, AP.VP.NR_VP(R2)
! # OF VP'S !
LD   R1, AP.READY_LIST(R2)
CHECK_RDONLY:
DO
I SEE IF READY LIST IS EMPTY!
CP R1, #NIL
IF EQ !LIST IS EMPTY!
THEN EXIT FROM CHECK_RDY_LIST
FI

CP APT.AP.STATE(R1), #RUNNING
IF EQ !PROCESS IS RUNNING!
THEN DON'T PREEMPT IT!

LD R5, APT.AP.VP_ID(R1)
!COMPUTE LOCATION IN PREEMPT VECTOR!
SUB R5, APT.VP.FIRST(R2)

LD R6, R15(R5)
LD @R6, #FALSE
ELSE PREEMPT IT!

INC R3
FI

DEC R4
CP R4, #0
IF EQ !ALL VP'S VERIFIED!
THEN

EXIT FROM CHECK_RDY_LIST
FI

!GET NEXT AP IN READY LIST!
LD R0, APT.AP.NEXT_AP(R1)
LD R1, R0
OD IEND CHECK_RDY_LIST!
!SET NECESSARY PREEMPTS!
LD R4, APT.VP.NR_VP(R2)
LD R1, APT.VP.FIRST(R2)
SEND_PREEMPT:
DO
POP R0, @R15
!CHECK TEMPLATE!
CP R0, #TRUE
IF EQ !CAN BE PREEMPTED!
THEN
CP R3, 0

IF GT IPREEMPTS REQUIRED!
THEN PREEMPT IT!

SAVE ARGUMENTS!
PUSH @R15, R1
PUSHL @R15, R2
PUSH @R15, R4
CALL SET_PREEMPT

IB1: VP IDI
! RESTORE ARGUMENTS!
POP R4, @R15
POP R2, @R15
POP R1, @R15
DEC R3

IF EQ!STACK RESTORED!
THEN
EXIT

EXIT

OD !END SEND_PREEMPT!
! CHECK NEXT READY LIST!
INC R2, #2
OD !END PREEMPT_CHECK!
! UNLOCK APT!
LDA R4, APT_LOCK
CALL K_UNLOCK

! RESTORE SUCCESS CODE!
LD R0, #SUCCEEDED

! RESTORE STACK!
ADD R15, #SIZEOF TEMP
RET
END TC_ADVANCE
TC_AWAIT  PROCEDURE

* CHECKS USER SPECIFIED VALUE *
* AGAINST CURRENT EVENTCOUNT *
* VALUE. IF USER VALUE IS LESS *
* THAN OR EQUAL EVENTCOUNT THEN *
* CONTROL IS RETURNED TO USER. *
* ELSE USER IS BLOCKED UNTIL *
* EVENT OCCURRENCE. *

* PARAMETERS: *
* R1: HANDLE POINTER *
* R2: INSTANCE (EVENT #) *
* RR4: SPECIFIED VALUE *

* RETURNS: *
* RO: SUCCESS CODE *

ENTRY

I ESTABLISH STACK FRAME FOR
TEMPORARY VARIABLES. !

SUB  R15, #SIZEOF TEMP

I SAVE INPUT PARAMETERS !

LD TEMP.HANDLE_PTR(R15), R1
LD TEMP.EVENT_NR(R15), R2
LDL TEMP.EVENT_VAL(R15), RR4

I LOCK APT !

LDA  R4, APT.LOCK
CALL K_LOCK

I RETURNS WHEN APT IS LOCKED !
I GET CURRENT EVENTCOUNT !

CALL MM_READ_EVENTCOUNT

!R1:HANDLE POINTER
R2:INSTANCE
RETURNS:
RO:SUCCESS_CODE
RR4:EVENTCOUNT!

CP  R0, #SUCCEEDED

IF EQ THEN

I DETERMINE IF REQUESTED EVENT
HAS OCCURRED !

LDL RR6, TEMP.EVENT_VAL(R15)
CPL RR6, RR4
IF GT !EVENT HAS NOT OCCURRED!
THEN !BLOCK PROCESS!

IDENTIFY PROCESS!
CALL RUNNING_VP !RETURNS:

SAVE RETURN VARIABLES!
LD TEMP.ID_VP(R15), R1
LD TEMP.CPU_NUM(R15), R3
LD R8, APT.RUNNING_LIST(R1)

RESTORE REMAINING ARGUMENTS!
LD R2, TEMP.EVENT_NR(R15)
LD R1, TEMP.HANDLE_PTR(R15)

SAVE EVENT DATA!
LDL RR4, HANDLE_VAL.HIGH(R1)
LDL APT.HANDLE(R8), RR4
LD R4, HANDLE_VAL.LOW(R1)
LD APT.HANDLE(2)(R8), R4
LD APT.INSTANCE(R8), R2
LD RR6, TEMP.EVENT_VAL(R15)
LD APT.VALUE(R8), RR6

REMOVE PROCESS FROM READY LIST!
LD R1, APT.AP.AFFINITY(R8)
LD R2, APT.READY_LIST(R1)

SEE IF PROCESS IS FIRST ENTRY IN READY LIST!
CP R2, R8
IF EQ [INSERT NEW READY LIST HEAD!]

LD R3, APT.AP.NEXT_AP(R3)
LD APT.READY_LIST(R1), R3
ELSE [DELETE FROM LIST BODY!]

DO R3, APT.AP.NEXT_AP(R2)
IF EQ !FOUND ITEM IN LIST!
THEN
LD R3, APT.AP.NEXT_AP(R8)
LD APT.AP.NEXT_AP(R2), R3
EXIT
FI
LD R2, R3
OD
FI
THREAD PROCESS IN BLOCKED LIST!
LD R2, R8
LDA R3, APT.BLOCKED_LIST
LDA R4, APT.AP.NEXT_AP
LDA R5, APT.AP_PRI
LDS R6, APT.AP.STATE
LDA R7, #BLOCKED
CALL LIST_INSERT IR2:OBJ ID
LD R1, TEMP.ID_VP(R15)
LD R3, TEMP.CPU_NUM(R15)
CALL TC_GETWORK IR1:VP_ID
LD R3, CPU #1
LDA R4, APT.LOCK
CALL K_UNLOCK
CALL K_UNLOCK
LD R0, #SUCCEEDED
FI
FI
RESTORE STACK!
- 283 -
0440 010F  ADD  R15, #SIZEOF TEMP
0442 0012  RET
0444 9E08  END TC_WAIT

- 284 -
PROCESS_CLASS  PROCEDURE

* READS SECURITY ACCESS *
* CLASS OF CURRENT PROCESS *
* IN APT. CALLED BY SEG *
* MGR AND EVENT MGR *

* LOCAL VARIABLES: *
* R1: VP ID *
* R5: PROCESS ID *

* RETURNS: *
* RR2: PROCESS SAC *

ENTRY

0446 7604  LDA  R4, APT.LOCK
0448 0000' CALL  K_LOCK  IB4:-APT.LOCK!
044A 5F00  CALL  RUNNING_VP  IRETURNS:
0450 0000*

    R1: VP_ID
    R3: CPU #1

0452 6115  LD  R5, APT.RUNNING_LIST(R1)
0454 0002'  LDL  RR2, APT.AP.SAC(R5)
0458 0024'  ! UNLOCK APT !

045A 7604  LDA  R4, APT.LOCK
045C 0000' CALL  K_UNLOCK
0460 0000'  RET
0462 9B08  END PROCESS_CLASS

- 285 -
GET_DBR_NUMBER PROCEDURE

* OBTAINS DBR NUMBER FROM APT *
* FOR THE CURRENT PROCESS. *
* CALLED BY SEGMENT MANAGER *

* LOCAL VARIABLES: *
* R1: VP ID *
* R5: PROCESS ID *

* RETURNS: *
* R1: DBR NUMBER *

ENTRY

!NOTE: DBR # IS ONLY VALID WHILE PROCESS IS LOADED. THIS IS NO PROBLEM IN SASS AS ALL PROCESSES REMAIN LOADED. IN A MORE GENERAL CASE, THE DBR # COULD ONLY BE ASSUMED CORRECT WHILE THE APT IS LOCKED!

0464 7604 LDA R4, APT.LOCK
0466 0000 CALL K_LOCK !R4:-APT.LOCK!
0468 5F00 CALL RUNNING_VP !RETURNS:
046A 0000* R1: VP_ID
046C 5F00 R3: CPU #
046E 0000* ! UNLOCK APT !
0470 6115 LD R5, APT.RUNNING_LIST(R1)
0472 0002 LD R1, APT.AP.DBR(R5)
0474 6151 0022 ! UNLOCK APT !
0476 7604 LDA R4, APT.LOCK
0478 0000 CALL K_UNLOCK
047A 5F00 CALL RUNNING_VP !RETURNS:
047C 0000* R1: VP ID
0480 9208 END GET_DBR_NUMBER
0482 END TC

END TC
Appendix C

DISTRIBUTED MEMORY MANAGER LISTINGS

Z8000ASM 2.02
LOC OBJ CODE STM T SOURCE STATEMENT

$LISTON $TTY
DIST_MM MODULE

CONSTANT

CREATE_CODE =: 50
DELETE_CODE =: 51
ACTIVATE_CODE =: 52
DEACTIVATE_CODE =: 53
SWAP_IN_CODE =: 54
SWAP_OUT_CODE =: 55
NR_CPU =: 2
NR_KST_ENTRY =: 54
MAX_SEG_SIZE =: 128
MAX_DBR_NO =: 4
KST_SEG_NO =: 2
NR_OF_KSEGS =: 10
BLOCK_SIZE =: 8
MEM_AVAIL =: %POO
G_AST_LIMIT =: 10
INSTANCE1 =: 1
INSTANCE2 =: 2
INVALID_INSTANCE =: 95
SUCCEEDED =: 2

TYPE

H_ARRAY ARRAY [3 WORD]
COM_HSG ARRAY [16 BYTE]
ADDRESS WORD

G_AST_REC RECORD
(UNIQUE_ID LONG
GLOBAL_ADDR ADDRESS
P_L_ASTE_NO WORD
FLAG WORD
PAR_ASTE WORD
NR_ACTIVE WORD
NO.ACT_DEP BYTE
SIZE1 BYTE

- 287 -
PG_TBL ADDRESS
ALIAS_TBL ADDRESS
SEQUENCER LONG
EVENT1 LONG
EVENT2 LONG
]

MM_VP_ID WORD

SEG_ARRAY ARRAY [MAX_SEG_SIZE BYTE]

$SECTION D_MM_DATA
GLOBAL

0000 MM_CPU_TBL ARRAY [NB_CPU MM_VP_ID]

$SECTION AVAIL_MEM
INTERNAL
! NOTE: MEM_POOL IS LOCATED IN CPU LOCAL MEMORY. !

0000 MEM_POOL ARRAY [MEM_AVAIL BYTE]

GLOBAL
! NOTE: NEXT_BLOCK IS USED IN THE MM_ALLOCATE STUB AS AN OFFSET POINTER INTO THE BLOCK OF ALLOCATABLE MEMORY. IT IS INITIALIZED IN BOOTSTRAP LOADER. !

0F00 NEXT_BLOCK WORD

$SECTION MSG_FRAME_DCL
INTERNAL
! NOTE: THESE RECORDS ARE "OVERLAYS" OR "FRAMES" USED TO DEFINE MESSAGE FORMATS. NO MEMORY IS ALLOCATED !

$ABS 0

0000 CREATE_MSG RECORD [ CR_CODE WORD
CE_MH_HANDLER H_ARRAY
CE_ENTRY_NO SHORT_INTEGER
CE_FILL BYTE
CE_SIZE WORD
CE_CLASS LONG]

$ABS 0

0000 DELETE_MSG RECORD [ DE_CODE WORD
DE_MH_HANDLER H_ARRAY
DE_ENTRY_NO SHORT_INTEGER
DE_FILL ARRAY[7 BYTE]]

$ABS 0

0000 ACTIVATE_MSG RECORD [ ACT_CODE WORD
A_DDR_NO WORD
A_MH_HANDLER H_ARRAY
A_ENTRY_NO SHORT_INTEGER
A_SEGMENT_NO SHORT_INTEGER
A_FILL LONG]
$ABS 0
DEACTIVATE_MSG RECORD [DEACT_CODE WORD
D_DBR_NO WORD
D_MM_HANDLE H_ARRAY
D_FILL ARRAY[ 3 WORD]]

$ABS 0
SWAP_IN_MSG RECORD [S_IN_CODE WORD
SI_MM_HANDLE H_ARRAY
SI_DBR_NO WORD
SI_ACCESS_AUTH BYTE
SI_FILL1 BYTE
SI.FILL ARRAY[ 2 WORD]]

$ABS 0
SWAP_OUT_MSG RECORD [S_OUT_CODE WORD
SO_DBR_NO WORD
SO_MM_HANDLE H_ARRAY
SO_FILL ARRAY[ 3 WORD]]

$ABS 0
RET_SUC_CODE RECORD[SUC_CODE BYTE
SC_FILL ARRAY[ 15 BYTE]]

$ABS 0
R_ACTIVATE_ARG RECORD [R_SUC_CODE BYTE
R_FILL BYTE
R_MM_HANDLE H_ARRAY
R_CLASS LONG
R_SIZE WORD
R_FILL1 WORD]

$ABS 0
MM_HANDLE RECORD
[ ID LONG
ENTRY_NO WORD
]

EXTERNAL
G_AST_LOCK WORD
G_AST ARRAY[G_AST_LIMIT G_AST_REC]
K_LOCK PROCEDURE
K_UNLOCK PROCEDURE
GET_CPU_NO PROCEDURE
SIGNAL PROCEDURE
WAIT PROCEDURE

- 289 -
GLOBAL
$SECTION D_MM_PROC

0000    MM_CREATE_ENTRY  PROCEDURE
*=======================================================================
* INTERFACE BETWEEN SEG MGR  *
* (CREATE_SEG PROCEDURE) AND  *
* MGR PROCESS (CREATE_ENTRY *
* PROCEDURE). ARRANGES AND  *
* PERFORMS IPC.  *
*=======================================================================
* REGISTER USE: *
* PARAMETERS *
* R0:SUCCESS_CODE (RET) *
* R1:HPTR (INPUT) *
* R2:ENTRY_NO (INPUT) *
* R3:SIZE (INPUT) *
* R4:CLASS (INPUT) *
* LOCAL USE *
* R6:MM_HANDLE ARRAY ENTRY *
* R8:-COM_MSGBUF *
* R13:-COM_MSGBUF *
*=======================================================================
ENTRY
!*USE STACK FOR MESSAGE!
0000 030F    SUB    R15,#SIZEOF COM_MSG
0002 0010    LD     R13,R15  ! -COM_MSGBUF !

!FILL COM_MSGBUF (LOAD MESSAGE). CREATE MSG FRAME IS BASED AT ADDRESS ZERO. IT IS OVERLaid ONTO COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADDING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
0006 4DD5    LD     CREATE_MSG.CR_CODE(R13),#CREATE_CODE
0008 0000
000A 0032
000C 3116    LD     R6,R1(#0)  !INDEX TO MM_HANDLE ENTRY!
000E 0000
0010 6FD6    LD     CREATE_MSG.CE_MM_HANDLE[0](R13),R6
0012 0002
0014 3116    LD     R6,R1(#2)
0016 0002
0018 6FD6    LD     CREATE_MSG.CE_MM_HANDLE[1](R13),R6
001A 0004
001C 3116    LD     R6,R1(#4)
001E 0004
0020 6FD6    LD     CREATE_MSG.CE_MM_HANDLE[2](R13),R6
0022 0006
0024 6FD2    LD     CREATE_MSG.CE_ENTRY_NO(R13),R2
0026 0008
0028 5DD4    LDL    CREATE_MSG.CE_CLASS(R13),R4
002A 000C

- 290 -
002C 6FD3   LD   CREATE_MSG.CE_SIZE(R13),R3
002E 000A   LD   R8,R13
0030 A1D8   CALL PERFORM_IPC 1R8: ~COM_MSGBUF!
0032 5P00   LD   R8,R13
0034 018C'  IRETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
0036 8D08   CLR    R0
0038 60D8   LDB   RLO,RET_SUC_CODE.SUC_CODE(R13)
003A 0000   ADD   R15,#SIZEOF COM_MSG  IRESTORE STACK STATE!
003C 010F   RET
0040 9E08   RET
0042       END MK_CREATE_ENTRY
MM_DELETE_ENTRY  PROCEDURE

* INTERFACE BETWEEN SEG MSG  *
* (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:~COM_MSGBUF  *
* R13:~COM_MSGBUF  *

ENTRY
IU$E STACK FOR MESSAGE!
SUB  R15,#SIZEOF COM_MSG

LD  R13,R15 !~COM_MSGBUF!

!FILL COM_MSGBUF (LOAD MESSAGE). DELETE_MSG FRAME
IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADDING
TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!

LD  DELETE_MSG.DE_CODE(R13),#DELETE_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.MM_HANDLE[1](R13),R6 
LD  R6,R1(#4) 
LD  DELETE_MSG.DE.MM_HANDLE[2](R13),R6 
LD  DELETE_MSG.DE_ENTRY_NO(R13),R2

LD  R8,R13 
CALL PERFORM_IPC  IR8: ~COM_MSGBUF!

!RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
CLR  R0 
LDB  RLO,RET_SUCC_CODE.SUCC_CODE(R13)
ADD  R15,#SIZEOF COM_MSG  !RESTORE STACK STATE!
RET

END MM_DELETE_ENTRY
PROGRAM ACTIVATE

* INTERFACE BETWEEN MGR * (MAKE_KOWN PROCEDURE) AND * MGR (ACTIVATE PROCEDURE). *
* ARRANGES AND PERFORMS IPC. *
* REGISTER USE: *
* PARAMETERS *
* R1: DBR_NO (INPUT) *
* R2: HPTR (INPUT) *
* R3: ENTRY_NO *
* R4: SEGMENT_NO *
* R12: HANDLE_PTR *
* LOCAL USE *
* R8: COMMSGBUF *
* R13: COMMSGBUF *
* RETURNS: *
* R0: SUCCESS CODE *
* R2: CLASS *
* R4: SIZE *
* USE STACK FOR MESSAGE *

ENTRY

SUB R15, SIZEOF COMMSG

LD R13, R15  
I -COMMSGBUF I
I SAVE RETURN HANDLE POINTER !

LD ACTIVATE_MSG.ACT_CODE(R13), #ACTIVATE_CODE

LD ACTIVATE_MSG.A_DBR_NO(R13), R1
LD R6, R2(#0)

LD ACTIVATE_MSG.AHANDLE[0](R13), R6
LD R6, R2(#2)

LD ACTIVATE_MSG.AHANDLE[1](R13), R6
LD R6, R2(#4)

LD ACTIVATE_MSG.AHANDLE[2](R13), R6
LD R6, R2(#0)

LD ACTIVATE_MSG.A_ENTRY_NO(R13), R3

- 293 -
LDB ACTIVATE_MSG.A_SEGMENT_NO(R13), R14
LD R8, R13
CALL PERFORM_IPC!(R8:~COM.MSGBUF!

I RESTORE RETURN HANDLE POINTER 
PPOP R12, @R15

I UPDATE MM_HANDLE ENTRY 
LDL R6, R_ACTIVATE_ARG.R_MM_HANDLE(R13)
LDL MM_HANDLE.ID(R12), R6

LD R6, R_ACTIVATE_ARG.R_MM_HANDLE[2](R13)

LD MM_HANDLE.ENTRY_NO(R12), R6

LD R1, R_ACTIVATE_ARG.R_SIZE(R13)
ADD R15, #SIZEOF COM_MSG !RESTORE STACK STATE!

END MM_ACTIVATE
**OODA**

```
MM_DEACTIVATE PROCEDURE
*********************************************
* INTERFACE BETWEEN SEG MGR             *
  *(TERMINATE PROCEDURE) AND            *
  * MNGR (DEACTIVATE PROCEDURE).        *
  * ARRANGES AND PERFORMS IPC.          *
*********************************************
* REGISTER USE:                          *
* PARAMETERS                              *
* R0:SUCCESS_CODE (RET)                  *
* R1:DBR_NO (INPUT)                      *
* R2:HPTE (INPUT)                        *
* LOCAL USE                               *
* R6:MM_HANDLE ARRAY ENTRY               *
* R8:-COM_MSGBUF                         *
* R13:-COM_MSGBUF                        *
*********************************************
ENTRY
USE STACK FOR MESSAGE!
00DA 030F SUB R15, #SIZEOF COM_MSG
00DC 0010 00DE A1FD LD R13, R15 # -COM_MSGBUF !

!FILL COM_MSGBUF (LOAD MESSAGE). DEACTIVATE_MSG FRAME IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADDING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!

00E0 4DD5 LD DEACTIVATE_MSG.DEACT_CODE(R13), #DEACTIVATE_CODE
00E4 0000 00E6 6FD1 LD DEACTIVATE_MSG.D_DBR_NO(R13), R1
00E8 0002 00EA 3126 LD R6, R2 (#0) !INDEX TO MM_HANDLE ENTRY!
00EC 0000 00EE 6FD6 LD DEACTIVATE_MSG.D_MM_HANDLE[0](R13), R6
00F0 0004 00F2 3126 LD R6, R2 (#2)
00F4 0002 00F6 6FD6 LD DEACTIVATE_MSG.D_MM_HANDLE[1](R13), R6
00F8 0006 00FA 3126 LD R6, R2 (#4)
00FC 0004 00FE 6FD6 LD DEACTIVATE_MSG.D_MM_HANDLE[2](R13), R6
0100 0008 0102 A1D8 LD R8, R13
0104 5F00 CALL PERFORM_IPC R8: -COM_MSGBUF !
0106 018C !

!RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
0108 8D08 CLR R0
010A 6D08 LDB R0, RET_SUC_CODE.SUC_CODE(R13)
```

- 295 -
ADD R15, #SIZEOF COM.MSG | RESTORE STACK STATE!

RET

END MM_DEACTIVATE

- 296 -
MM_SWAP_IN  PROCEDURE

*****************************************************************************
* INTERFACE BETWEEN SEG MGR (SM_*
* SWAP_IN PROCEDURE) AND MGR *
* (SWAP_IN PROCEDURE). ARRANGES *
* AND PERFORMS IPC. *
*****************************************************************************
* REGISTER USE: *
* PARAMETERS *
* R0: SUCCESS_CODE (RET) *
* R1: DBR_NO (INPUT) *
* R2: HPTR (INPUT) *
* R3: ACCESS (INPUT) *
* LOCAL USE *
* R6: MM_HANDLE ARRAY ENTRY *
* R8: -COM_MSGBUF *
* R13: -COM_MSGBUF *
*****************************************************************************
ENTRY
IUSE STACK FOR MESSAGE!
0114 030F  SUB  R15,*SIZEOF COM_MSG
0116 0010  LD  R13,R15  !-COM_MSGBUF!

!FILL COM_MSGBUF (LOAD MESSAGE). SWAP_IN_MSG FRAME IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADDING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!

011A 4DD5  LD  SWAP_IN_MSG.S_IN_CODE(R13),#SWAP_IN_CODE
011C 0000
011E 0036
0120 3126  LD  R6,R2(#0)  INDEX TO MM_HANDLE ENTRY!
0122 0000
0124 6FD6  LD  SWAP_IN_MSG.SI_MM_HANDLE[0](R13),R6
0126 0002
0128 3126  LD  R6,R2(#2)
012A 0002
012C 6FD6  LD  SWAP_IN_MSG.SI_MM_HANDLE[1](R13),R6
012E 0004
0130 3126  LD  R6,R2(#4)
0132 0004
0134 6FD6  LD  SWAP_IN_MSG.SI_MM_HANDLE[2](R13),R6
0136 0006
0138 6FD1  LD  SWAP_IN_MSG.SI_DBR_NO(R13),R1
013A 0008
013C 6EDB  LDB  SWAP_IN_MSG.SI_ACCESS_AUTH(R13),R13
013E 000A
0140 A1D8  LD  R8,R13
0142 5FE0  CALL  PERFORM_IPC  !R8: -COM_MSGBUF!
0144 018C  !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
0146 8D08  CLR  R0
0148 60D8  LDB  RLO,RET_SUC_CODE,SUC_CODE(R13)
014A 0000
014C 010F ADD R15,#SIZEOF COM.MSG IRESTORE STACK STATE!
014E 0010
0150 9E08 RET
0152 END MM_SWAP_IN
**MM_SWAP_OUT**  
PROCEDURE

* INTERFACE BETWEEN SSG_MGR (SM_*)  
* SWAP_OUT_PROCEDURE) AND MGR  
* (SWAP_OUT_PROCEDURE). ARRANGES*  
* AND PERFORMS IPC.  

* REGISTER USE:  
* PARAMETERS  
* B0: SUCCESS_CODE (RET)  
* R1: DBR_NO (INPUT)  
* R2: HPTR (INPUT)  
* LOCAL USE  
* R6: MM_HANDLE ARRAY ENTRY  
* R8: COMJISGBUF  
* R13: COM_MSGBUF  

ENTRY

USE STACK FOR MESSAGE!

! USE STACK FOR MESSAGE!

! USE STACK FOR MESSAGE!

!!! USE STACK FOR MESSAGE!!!

**MM_SWAP_OUT**

! INTERFACE BETWEEN SSG_MGR (SM_*)

* SWAP_OUT_PROCEDURE) AND MGR  
* (SWAP_OUT_PROCEDURE). ARRANGES*

* AND PERFORMS IPC.  

* REGISTER USE:  
* PARAMETERS  
* B0: SUCCESS_CODE (RET)  
* R1: DBR_NO (INPUT)  
* R2: HPTR (INPUT)  

* LOCAL USE  
* R6: MM_HANDLE ARRAY ENTRY  
* R8: COMJISGBUF  
* R13: COM_MSGBUF  

ENTRY

USE STACK FOR MESSAGE!

SUB R15, #SIZEOF COM_MSG

LD R13, R15 - COM_MSGBUF

LD RD(0), R6, R2  
INDEX TO MM Handle ENTRY!

LD SWAP_OUT_MSG.SO_MM_HANDLE[0](R13), 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_HANDLE[2](R13), R6

LD SWAP_OUT_MSG.SO_DBR_NO(R13), R1

LD R8, R13

CALL PERFORM_IPC R6: COM_MSGBUF

! RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!

CLR R0

LD R0, RET_SUC_CODE.SUC_CODE(R13)

ADD R15, #SIZEOF COM_MSG 
[RESTORE STACK STATE!]

- 299 -
PERFORM_IPC

PROCEDURE

SERVICE ROUTINE TO ARRANGE AND PERFORM IPC WITH THE MEM MGR PROC

* REGISTER USE:
* PARAMETERS
* R8: ~COM_MSG(INPUT)
* LOCAL USE
* R1, R2: WORK REGS
* R4: ~G_AST_LOCK
* R13: ~COM_MSGBUF

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

ENTRY

018C 93FD PUSH @R15,R13 1~COM_MSGBUF!
018E 5F00 CALL GET_CPU_NO !RET-R1:CPU_NO!
0190 0000* 0192 A112 LD R2,R1
0194 6121 LD R1, MM_CPU_TBL(R2) IMM_VP_ID!
0196 0000* 0198 7604 LDA R4,G_AST_LOCK
019A 0000* 019C 5F00 CALL K_LOCK
019E 0000* 01A0 5F00 CALL SIGNAL 1R1:MM_VP_ID,R8:~COM_MSGBUF!
01A2 0000* 01A4 97FD POP R13,@R15
01A6 A1DB LD R8,R13 1~COM_MSGBUF!
01A8 93FD PUSH @R15,R13
01AA 5F00 CALL WAIT 1R8:~COM_MSGBUF!
01AC 0000* 01AE 7604 LDA R4,G_AST_LOCK
01B0 0000* 01B2 5F00 CALL K_UNLOCK
01B4 0000* 01B6 97FD POP R13,@R15
01B8 9B08 RET
01BA END PERFORM_IPC
MM_ALLOCATE PROCEDURE

* ALLOCATES BLOCKS OF CPU LOCAL MEMORY. EACH BLOCK CONTAINS 256 BYTES OF MEMORY.

PARAMETERS:
* R3: # OF BLOCKS
* RETURNS:
* R2: STARTING ADDR
* LOCAL:
* R4: BLOCK POINTER

ENTRY

NOTE: THIS PROCEDURE IS ONLY A STUB OF THE ORIGINALLY DESIGNED MEMORY ALLOCATING MECHANISM. IT IS USED BY THE PROCESS MANAGEMENT DEMONSTRATION TO ALLOCATE CPU LOCAL MEMORY FOR ALL MEMORY ALLOCATION REQUIREMENTS. IN AN ACTUAL SASS ENVIRONMENT, THIS WOULD BE BETTER SERVED TO HAVE SEPARATE ALLOCATION PROCEDURES FOR KERNEL AND SUPERVISOR NEEDS. (E.G., KERNEL_ALLOCATE AND SUPERVISOR_ALLOCATE).

I Compute size of memory requested
SLL R3, #BLOCK_SIZE

I Compute offset of memory that is to be allocated
LD R4, NEXT_BLOCK !OFFSET!

I Update offset in section of available memory to indicate that currently requested memory is now allocated
LD NEXT_BLOCK, R4 I SAVE OFFSET!
**MM_TICKET PROCEDURE**

* RETURNS CURRENT VALUE OF *
* SEGMENT SEQUENCER AND *
* INCREMENTS SEQUENCER VALUE* *
* FOR NEXT TICKET OPERATION *

**PARAMETERS:**
* R1: SEG HANDLE PTR *
* RETURNS: *
* RR4: TICKET VALUE *
* LOCAL VARIABLES: *
* RR6: SEQUENCER VALUE *
* R8: G_AST ENTRY # *

**ENTRY**

```
01CE 93F1  PUSH     @R15, R1
01D0 7604  LDA      R4, G_AST_LOCK
01D2 0000* CALL     K_LOCK
01D4 5F00
01D6 0000*
01D8 97F1  POP      R1, @R15
01DA 6118  LD       R8, MM_HANDLE.ENTRYNO(R1)
01DC 0004
01DE 5486  LDL      RR6, G_AST.SEQUENCER(R8)
01E0 0014*
01E2 9464  LDL      RR4, RR6
01E4 1606  ADDL     RR6, #1
01E6 0000
01E8 0001
01EA 5D86  LDL      G_AST.SEQUENCER(R8), RR6
01EC 0014*  
01EE 91F4  PUSHL    @R15, RR4
01FO 7604  LDA      R4, G_AST_LOCK
01F2 0000* CALL     K_UNLOCK
01F4 5F00
01F6 0000*
01F8 95F4  POPL     RR4, @R15
01FA 9208  RET
01FC  END  MM_TICKET
```

---

* - 303 -
**MN_READ_EVENTCOUNT**  **PROCEDURE**

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
0234 5E08 ELSE !INVALID INPUT!
0236 023C' LD R0, #INVALID_INSTANCE
0238 2100 INVALID INSTANCE
023A 005F FI
NOT: NO VALUE IS RETURNED IF USER SPECIFIED INVALID EVENT #1
SAVE RETURN VALUES!
023C 91F4 PUSHL $R15, %R4
UNLOCK G_AST I
023E 7604 LDA %R4, G_AST_LOCK
0240 0000* CALL F_UNLOCK
0242 5F00 I RESTORE RETURN VALUES I
0244 0000* POPL %R4, $R15
0246 95F4 POPL %R4, $R15
0248 9E08 RET
024A END MM_READ_EVENTCOUNT
MM_ADVANCE  PROCEDURE

* DETERMINES G_AST OFFSET FROM  *
* SEGMENT HANDLE AND INCREMENTS  *
* THE INSTANCE(EVENT #) SPECIFIED  *
* BY THE CALLER. THIS IN EFFECT  *
* ANNOUNCES THE OCCURRENCE OF THE  *
* EVENT. THE NEW VALUE OF THE  *
* EVENTCOUNT IS RETURNED TO THE  *
* CALLER.  *

PARAMETERS:
* R1: HANDLE POINTER
* R2: INSTANCE(EVENT #)

RETURNS:
* RR2: NEW EVENTCOUNT VALUE

ENTRY
! SAVE INPUT PARAMETERS!

024A 93F1  PUSH @R15, R1
024C 93F2  PUSH @R15, R2
! LOCK G_AST!
024E 7604  LDA R4, G_AST_LOCK
0250 0000*  CALL K_LOCK
0252 5E00  CALL K_LOCK
0254 0000*  ! RESTORE INPUT PARAMETERS!
0256 97F2  POP R2, @R15
0258 97F1  POP R1, @R15
! GET G_AST OFFSET!
025A 6114  LD R4, MM_HANDLE_ENTRY_NO(R1)
025C 0004

! DETERMINE INSTANCE!
IF R2
025E 0B02  CASE #INSTANCE1 THEN
0260 0011
0262 5E0E
0264 027C*
0266 5442  LDL RR2, G_AST_EVENT1(R4)
0268 0018*  ADDL RR2, #1
026A 1602
026C 0000
026E 0001  ! SAVE NEW EVENTCOUNT!
0270 5D42  LDL G_AST_EVENT1(R4), RR2
0272 0018*  ADDL RR2, #1
0274 2100  LD R0, #SUCCEEDED
0276 0002
0278 5E08  CASE #INSTANCE2 THEN
027A 029E*  ADDL RR2, #1
027C 0B02
027E 0002  ! END OF ROUTINE!
0280  5E0E
0282  029A
0284  5A42
0286  001C*
0288  1602
028A  0000
028C  0001
028E  5D42
0290  001C*
0292  2100
0294  0002
0296  5E08
0298  029E*
029A  2100
029C  005F

LDL  RR2,  G_AST.EVENT2(R4)
ADDL  RR2,  #1

! SAVE NEW EVENTCOUNT !
LDL  G_AST.EVENT2(R4),  RR2

! NOTE: AN INVALID INSTANCE VALUE WILL NOT AFFECT EVENT DATA !
! UNLOCK G_AST !

LDA  R4,  G_AST_LOCK
CALL  K_UNLOCK

END  MM.Advance
END  DIST_MM
Appendix D

GATE_KEEPER LISTINGS

Z8000ASM 2.02

LOC OBJ CODE STMT SOURCE STATEMENT

KERNEL_GATE_KEEPER MODULE

$LISTON $TTY

CONSTANT
ADVANCE_CALL := 1
AWAIT_CALL := 2
CREATE_SEG_CALL := 3
DELETE_SEG_CALL := 4
MAKEKNOWN_CALL := 5
READ_CALL := 6
SM_SWAP_IN_CALL := 7
SM_SWAP_OUT_CALL := 8
TERMINATE_CALL := 9
TICKET_CALL := 10
WRITE_CALL := 11
WRITELN_CALL := 12
CRLF_CALL := 13
WRITE := %0FC8 !PRINT CHAR!
WRITELN := %0FC0 !PRINT MSG!
CRLF := %0FD4 !CAR RET/LINE FEED!
MONITOR := %A902
REGISTER_BLOCK := 32
TRAP_CODE_OFFSET := 36
INVALID_KERNEL_ENTRY := %BAD

GLOBAL
GATE_KEEPER_ENTRY LABEL

EXTERNAL
ADVANCE PROCEDURE
AWAIT PROCEDURE
CREATE_SEG PROCEDURE
DELETE_SEG PROCEDURE
MAKEKNOWN PROCEDURE
READ PROCEDURE
SM_SWAP_IN PROCEDURE
SM_SWAP_OUT PROCEDURE
TERMINATE PROCEDURE
TICKET PROCEDURE
KERNEL_EXIT LABEL

- 308 -
INTERNAL
$SECTION KERNEL_\_GATE_\_PROC

0000 GATE_\_KEEPER_\_MAIN \_PROCEDURE

ENTRY
GATE_\_KEEPER_\_ENTRY:
   | SAVE REGISTERS !
0000 030F  SUB  R15, #REGISTER_\_BLOCK
0002 0020
0004 1C99  LDM  @R15, R1, #16
0006 010F  | SAVE NSP !
0008 9392  PUSH  @R15, R2
000A 7D27  LDCTL R2, NSP
   | RESTORE INPUT REGISTERS !
000C 2D92  BX  R2, @R15
   | SAVE REGISTER 2 !
000E 9392  PUSH  @R15, R2
   | GET SYSTEM TRAP CODE !
0010 3192  LD  R2, R15(#TRAP_\_CODE_OFFSET)
0012 0024  | REMOVE SYSTEM CALL IDENTIFIER FROM SYSTEM TRAP INSTRUCTION !
0014 8C28  CLRB RH2
   | NOTE: THIS LEAVES THE USER VISIBLE
   | EXTENDED INSTRUCTION NUMBER IN R2 !
   | DECODE AND EXECUTE EXTENDED INSTRUCTION !
   | NOTE: THE INITIAL VALUE FOR REGISTER 2
   | WILL BE RESTORED WHEN THE APPROPRIATE
   | CONDITION IS FOUND !
0016 0B02  CASE  #ADVANCE_\_CALL THEN
0018 0001
001A 5E02
001C 0028* 001E 97F2  POP  R2, @R15
0020 5F90  CALL ADVANCE
0022 0000* 0024 5E08  CASE  #WAIT_\_CALL THEN
0026 010C* 0028 9B02
002A 0002 002C 5E08 002E 003A* 0030 97F2  POP  R2, @R15
0032 5F90  CALL WAIT
0034 0000* 0036 5E08  CASE  #CREATE_\_SEG_\_CALL THEN
0038 010C* 003A 0B02
003C 0003 003E 5E08

- 309 -
0040 0C  
0042 97F2  
0044 5F00  
0046 0000*  
0048 5E08  
004A 010C*  
004C 0B02  
004E 0004  
0050 5E0E  
0052 0070*  
0054 97F2  
0056 5F00  
0058 0000*  
005A 5E08  
005C 010C*  
005E 0B02  
0060 0005  
0062 5E0E  
0064 0070*  
0066 97F2  
0068 5F00  
006A 0000*  
006C 5E08  
006E 010C*  
0070 0B02  
0072 0006  
0074 5E0E  
0076 0082*  
0078 97F2  
007A 5F00  
007C 0000*  
007E 5E08  
0080 010C*  
0082 0B02  
0084 0007  
0086 5E0E  
0088 0094*  
008A 97F2  
008C 5F00  
008E 0000*  
0090 5E08  
0092 010C*  
0094 0B02  
0096 0008  
0098 5E0E  
009A 00A6*  
009C 97F2  
009E 5F00  
00A0 0000*  
00A2 5E08  
00A4 010C*  
00A6 0B02  
00A8 0009  
00AA 5E0E

- 310 -
00AC 00B8'  POP  R2, &R15
00AE 97F2  CALL TERMINATE
00B0 5F00  CALL TICKET
00B4 5E08  CASE #TICKET_CALL THEN
00B6 010C'  POP  R2, &R15
00B8 0802
00BA 000A
00BC 5E0E
00BE 00CA'
00C0 97F2  POP  R2, &R15
00C2 5F00  CALL WRITE
00C4 5101
00C6 5E08  CASE #WRITE_CALL THEN
00C8 010C'
00CA 0802
00CC 000B
00CE 5E0E
00D0 000C'
00D2 97F2  POP  R2, &R15
00D4 5F00  CALL WRITELN
00D6 0FC8
00D8 5E08  CASE #WRITELN_CALL THEN
00DA 010C'
00DC 0802
00DE 000C
00E0 5E0E
00E2 00EE'
00E4 97F2  POP  R2, &R15
00E6 5F00  CALL WRITELN
00E8 0FC0
00EA 5E08  CASE #CRLF_CALL THEN
00EC 010C'
00EE 0802
00F0 000D
00F2 5E0E
00F4 0100'
00F6 97F2  POP  R2, &R15
00F8 5F00  CALL CRLF
00FA 0FD4
00FC 5E08  ELSE: INVALID KERNEL INVOCATION!
00FE 010C'

I RETURN TO MONITOR!
I NOTE: THIS RETURN TO MONITOR IS
FOR STUB USE ONLY. AN INVALID
KERNEL INVOCATION WOULD NORMALLY
RETURN TO USER. !

100 7601  LDA  R1, 5
1002 0100'  LD  R0, #INVALID_KERNEL_ENTRY
1004 2100
1006 09AD
1008 5F00  CALL MONITOR
100A A902
FI
I SAVE REGISTERS ON KERNEL STACK!
I SAVE R1!
010C 93F1
PUSH @R15, R1
I GET ADDRESS OF REGISTER BLOCK!
010E 34F1
LDA R1, R15(#4)
0110 0004
I SAVE REGISTERS IN REGISTER BLOCK
ON KERNEL STACK.
0112 1C19
LDM @R1, R1, #16
0114 010F
I RESTORE R1 BUT MAINTAIN ADDRESS
OF REGISTER BLOCK!
0116 2DF1
EX R1, @R15
I SAVE R1 ON STACK!
0118 33F1
LD R15(#4), R1
011A 0004
I RESTORE REGISTER BLOCK ADDRESS!
011C 97F1
POP R1, @R15
I SAVE VALID EXIT SP VALUE!
011E 33F1
LD R15(#30), R1
0120 001E
I EXIT KERNEL BY MEANS OF HARDWARE
PREEMPT HANDLER!
0122 5E08
JP KERNEL_EXIT
0124 0000*
0126
END GATE_KEEPER_MAIN
END KERNEL_GATE_KEEPER
USER_GATE            MODULE

$LISTON $TTY

CONSTANT

ADVANCE_CALL  :  1
AWAIT_CALL    :  2
CREATESEG_CALL:  3
DELETESEG_CALL:  4
MAKE KNOWN_CALL:  5
READ_CALL     :  6
SM_SWAP_IN_CALL:  7
SM_SWAP_OUT_CALL:  8
TERMINATE_CALL:  9
WRITE_CALL    : 10
WRITELN_CALL  : 12
CRLF_CALL     : 13

GLOBAL
$SECTION USER_GATE_PROC

0000  ADVANCE      PROCEDURE
!***************************
*   PARAMETERS: *
*   R1:SEGMENT # *
*   R2:INSTANCE (ENTRY#)*
***************************
*   RETURNS: *
*   R0:SUCCESS CODE *
***************************!
ENTRY
0000 7F01   SC  #ADVANCE_CALL
0002 9E08   RET
0004  END ADVANCE

0004  AWAIT       PROCEDURE
!***************************
*   PARAMETERS: *
*   R1:SEGMENT # *
*   R2:INSTANCE *
*   RR4:SPECIFIED VALUE *
***************************
*   RETURNS: *
*   R0:SUCCESS CODE *
***************************!
ENTRY
0004 7F02   SC  #AWAIT_CALL
0006 9E08   RET
0008  END AWAIT

- 313 -
CREATE_SEG procedure

PARAMETERS:
* R1:MENTOR_SEG_NO *
* R2:ENTRY_NO *
* R3:SIZE *
* R4:CLASS *

RETURNS:
* R0:SUCCESS CODE *

ENTRY

CREATE_SEG_CALL

END CREATE_SEG

DELETE_SEG procedure

PARAMETERS:
* R1:MENTOR_SEG_NO *
* R2:ENTRY_NO *

RETURNS:
* R0:SUCCESS CODE *

ENTRY

DELETE_SEG_CALL

END DELETE_SEG

MAKEKNOWN procedure

PARAMETERS:
* R1:MENTOR_SEG_NO *
* R2:ENTRY_NO *
* R3:ACCESS DESIRED *

RETURNS:
* R0:SUCCESS CODE *
* R1:SEGMENT # *
* R2:ACCESS ALLOWED *

ENTRY

MAKEKNOWN_CALL

END MAKEKNOWN

READ procedure

PARAMETERS:
* R1:SEGMENT # *
* R2:INSTANCE *

END READ

END - 314 -
* RETURNS: *
* RO: SUCCESS CODE *
* RR4: EVENTCOUNT *
**************************************************************************
ENTRY
0014 7F06 SC #READ_CALL
0016 9E08 RET
0018 END READ

0018 SM_SWAP_IN PROCEDURE
**************************************************************************
* PARAMETERS: *
* R1: SEGMENT # *
**************************************************************************
ENTRY
0018 7F07 SC #SM_SWAP_IN_CALL
001A 9E08 RET
001C END SM_SWAP_IN

001C SM_SWAP_OUT PROCEDURE
**************************************************************************
* PARAMETERS: *
* R1: SEGMENT # *
**************************************************************************
ENTRY
001C 7F08 SC #SM_SWAP_OUT_CALL
001E 9E08 RET
0020 END SM_SWAP_OUT

0020 TERMINATE PROCEDURE
**************************************************************************
* PARAMETERS: *
* R1: SEGMENT # *
**************************************************************************
ENTRY
0020 7F09 SC #TERMINATE_CALL
0022 9E08 RET
0024 END TERMINATE

0024 TICKET PROCEDURE
**************************************************************************
* PARAMETERS: *
* R1: SEGMENT # *
**************************************************************************
* RO:SUCCESS CODE *
* RR4:TICKET VALUE *
******************************************************************************
ENTRY
0024 7F0A SC #TICKET_CALL
0026 9E08 RET
0028 END TICKET
0028 WRITE PROCEDURE
ENTRY
0028 7F0B SC #WRITE_CALL
002A 9E08 RET
002C END WRITE
002C WRITELN PROCEDURE
ENTRY
002C 7F0C SC #WRITELN_CALL
002E 9E08 RET
0030 END WRITELN
0030 CRLF PROCEDURE
ENTRY
0030 7F0D SC #CRLF_CALL
0032 9E08 RET
0034 END CRLF
Appendix E

BOOTSTRAP_LOADER LISTINGS

Z8000ASM 2.02
LOC OBJ CODE STMT SOURCE STATEMENT

BOOTSTRAP_LOADER MODULE

$LISTON $TTY
CONSTANT

! ******** SYSTEM PARAMETERS ********
NR_CPU := 2
NR_VP := NR_CPU*4
NR_AVAL_VP := NR_CPU*2
MAX_DBG_NR := 10
STACK_SEG := 1
STACK_SEG_SIZE := %100
STACK_BLOCK := STACK_SEG_SIZE/256

! * * OFFSETS IN STACK SEG * * !
STACK_BASE := STACK_SEG_SIZE-%10
STATUS_REG_BLOCK := STACK_SEG_SIZE-%10
INTERRUPT_FRAME := STACK_BASE-4
INTERRUPT_REG := INTERRUPT_FRAME-34
N_S_P := INTERRUPT_REG-2
F_C_W := STACK_SEG_SIZE-%E

! ******** SYSTEM CONSTANTS ********
ON ::= %FFFF
OFF ::= 0
READY ::= 1
NIL ::= %FFFF
INVALID ::= %EEEE
KERNEL_PCW ::= %5000
AVAILABLE ::= 0
ALLOCATED ::= %FF
SC_OFFSET ::= 12

TYPE

MESSAGE ARRAY [16 BYTE]
ADDRESS WORD
MR_VP_ID WORD
VP_INDEX INTEGER
MSG_INDEX INTEGER

- 317 -
MSG_TABLE RECORD
[ MSG MESSAGE
  SENDER VP_INDEX
  NEXT_MSG MSG_INDEX
  FILLER ARRAY [6, WORD]
]

VP_TABLE RECORD
[ DBR ADDRESS
  PRI WORD
  STATE WORD
  IDLE_FLAG WORD
  PREEMPT WORD
  PHYS_PROCESSOR WORD
  NEXT_READY_VP VP_INDEX
  MSG_LIST MSG_INDEX
  EXT_ID WORD
  FILLER_1 ARRAY [7, WORD]
]

EXTERNAL
GET_DBR_ADDR PROCEDURE
CREATE_STACK PROCEDURE
LIST_INSERT PROCEDURE
ALLOCATE_MNU PROCEDURE
UPDATE_MNU_IMAGE PROCEDURE
MM_ALLOCATE PROCEDURE
MM_ENTRY LABEL
IDLE_ENTRY LABEL
PREEMPT_RET LABEL
BOOTSTRAP_ENTRY LABEL
GATE_KEEPER_ENTRY LABEL
NEXT_BLOCK WORD
MM_CPU_TBL ARRAY [NR_CPU MM_VP_ID]

VPT TABLE RECORD
[ LOCK WORD
  RUNNING_LIST ARRAY [NR_CPU WORD]
  READY_LIST ARRAY [NR_CPU WORD]
  FREE_LIST MSG_INDEX
  VIRT_INT_VEC ARRAY [1, ADDRESS]
  FILLER_2 WORD
  VP ARRAY [NR_VP, VP_TABLE]
  MSG_Q ARRAY [NR_VP, MSG_TABLE]
]
EIT_VP_LIST ARRAY[NR_AVAIL_VP WORD]
NEXT_AVAIL_MNU ARRAY[MAX_DRV_NR BYTE]

PRDS RECORD
[PHYS_CPU_ID WORD
LOG_CPU_ID INTEGER
VP_NR WORD
IDLE_VP VP_INDEX]

INTERNAL
$SECTION LOADER_DATA

! NOTE: THESE DECLARATIONS WILL NOT WORK IN A TRUE MULTIPROCESSOR ENVIRONMENT AS THEY ARE SUBJECT TO A "CALL." THEY MUST BE DECLARED AS A SHARED GLOBAL DATABASE WITH "RACE" PROTECTION (E.G., LOCK).

0000 NEXT_AVAIL_VP INTEGER
0002 NEXT_EXT_VP INTEGER
$SECTION  LOADER_INT
INTERNAL

BOOTSTRAP  PROCEDURE

**********************************************************
* CREATES KERNEL PROCESSES AND *
* INITIALIZES KERNEL DATABASES.*
* INCLUDES INITIALIZATION OF *
* VIRTUAL PROCESSOR TABLE, *
* EXTERNAL VP LIST, AND MMU *
* IMAGES.  ALLOCATES MMU IMAGE *
* AND CREATES KERNEL DOMAIN *
* STACK FOR KERNEL PROCESSES. *
**********************************************************

ENTRY
!
! INITIALIZE PRDS AND MMU POINTER !
! NOTE: THE FOLLOWING CONSTANTS ARE
ONLY TO BE INITIALIZED ONCE. THIS
WILL OCCUR DURING SYSTEM INITIALIZATION!

0000 4D05   LD     PRDS.PHYS_CPU_ID, $FFFF
0002 0000*
0004 FFFF

! NOTE: LOGICAL CPU NUMBERS ARE ASSIGNED
IN INCREMENTS OF 2 TO FACILITATE INDEXING
(OFFSETS) INTO LISTS SUBSCRIPTED BY
LOGICAL CPU NUMBER. !

0006 4D05   LD     PRDS.LOG_CPU_ID, #2
0008 0002*
000A 0002

! SPECIFY NUMBER OF VIRTUAL PROCESSORS
ASSOCIATED WITH PHYSICAL CPU. !

000C 4D05   LD     PRDS.VP_NR, #2
000E 0004*
0010 0002
0012 4D08   CLR   NEXT_BLOCK
0014 0000*
0016 4D08   CLR   NEXT_AVAIL_vp
0018 0000*
001A 4D08   CLR   NEXT_EXT_vp
001C 0002*

! ESTABLISH GATE KEEPER AS SYSTEM CALL
TRAP HANDLER !
! GET BASE OF PROGRAM STATUS AREA !

001E 7D15   LDCTL  R1, PSAP

! ADD SYSTEM CALL OFFSET TO PSA BASE ADDR !

0020 0101   ADD   R1, #SC_OFFSET
0022 000C

! STORE KERNEL FCW IN PSA !

0024 0D15   LD     OR1, #KERNEL_FCW
0026 5000

! STORE ADDRESS OF GATE KEEPER IN PROGRAM
STATUS AREA AS SYSTEM TRAP HANDLER !

0028 A911   INC   R1, #2

- 320 -
002A  0D15  LD  @R1, #GATE_KEEPER_ENTRY
002C  0000* CLR  R1 ! NEXT_AVAIL_MMU_INDEX !

! INITIALIZE ALL MMU IMAGES AS AVAILABLE !

SET_MMU_MAP:

DO

0030  4C15  LDB  NEXT_AVAIL_MMU(R1), #AVAILABLE
0032  0000*  
0034  0000  
0036  A910  INC  R1, #1
! CHECK FOR END OF TABLE !
0038  0B01  CP  R1, #MAX_DBR_NR
003A  000A  
003C  5E00  IF EQ THEN EXIT FROM SET_MMU_MAP FI
003E  0004*  
0040  5E08  
0042  0046*  
0044  E8F5  OD

! CREATE MEMORY MANAGER PROCESS !

0046  2103  LD  R3, #STACK_BLOCK
0048  0001  

! ALLOCATE AND INITIALIZE KERNEL
DOMAIN STACK SEGMENT !

004A  5F00  CALL  MM_ALLOCATE  183: # OF BLOCKS
004C  0000*  

    RETURNS
    R2: START ADDR!

004E  A121  LD  R1, R2
0050  2103  LD  R3, #KERNEL_FCN
0052  5E00  
0054  7604  LDA  R4, MM_ENTRY
0056  0000*  
0058  6105  LD  R5, #FPFFInsn!
005A  FFF0  
005C  7606  LDA  R6, PREEMPT_RET
005E  0000*  
0060  93F1  PUSH  @R15, R1 !SAVE STACK ADDR!
0062  030F  SUB  R15, #8
0064  0008  
0066  1CF9  LDM  @R15, R3, #4
0068  0303  
006A  A1F0  LD  R0, R15

! NOTE: ARGLIST FOR CREATE STACK INCLUDES
KERNEL_FCN, INITIAL IC, WSP, AND INITIAL
RETURN POINT. !

006C  5F00  CALL  CREATE_STACK  ! (R0: ARGUMENT PTR
006E  0000*  

R1: TOP OF STACK
R2-R14: INITIAL
REG.STATES !
ADD    R15, #8  IOVERLAY ARGUMENTS!

CALL  ALLOCATE_MNU  IRETURNS:

LD     R1, #STACK_SEG  I SEGMENT NO. 1

POP    R2, @R15  IGET STACK ADDR

LD     R3, #0  I WRITE ATTRIBUTE

CALL  ALLOCATE_MNU  I(RO: DBR #)

LD     R4, #STACK_BLOCK-1

PUSH   @R15, RO

CALL  UPDATE_MNU_IMAGE  I(RO: DBR #)

R1: SEGMENT #
R2: SEG ADDRESS
R3: SEG ATTRIBUTES
R4: SEG LIMITS)

PUSH   RO, @R15

CALL  GET_DBR_ADDR  I(RO: DBR #)

R1: DBR ADDRESS)

LD     R2, #2  I PRIORITY

LD     R5, #OFF  I PREEMPT

LD     R6, #OFF  I KERNEL PROCESS

CALL  UPDATE_VPTABLE  I(R1: DBR

R2: PRIORITY
R5: PREEMPT FLAG
R6: EXT_VP FLAG)

R9: VP_ID )

CALL  UPDATE_VPTABLE  I(R1: DBR

R2: PRIORITY
R5: PREEMPT FLAG
R6: EXT_VP FLAG)

R9: VP_ID )

LD     R10, PRDS.LOG_CPU_ID

- 322 -
00A6 6FA9  LD MM_CPU_TBL(R10), R9
00A8 0000*  CREATE IDLE PROCESS!
00AA 2103  LD R3, #STACK_BLOCK
00AC 0001  ! OF BLOCKS
00AE 5F00  CALL MM_ALLOCATE(R3): RETURNS
00B0 0000*  R2: START ADDRESS

00B2 A121  LD R1, R2
00B4 2103  LD R3, #KERNEL_FCW
00B6 5000  !
00B8 7604  LDA R4, IDLE_ENTRY
00BA 0000*  !
00BC 2105  LD R5, #PPP! INST!
00BE 7606  LDA R6, PREEMPT_SET
00C0 93F1  PUSH @R15, R1 !SAVE STACK ADDRESS
00C2 0000*  !
00C4 030F  SUB R15, #8 !INITIAL REG. STATES!
00C6 0008  !
00C8 1CF9  LDM @R15, R3, #4
00CA 0303  !
00CE A1F0  LD R0, R15

! INITIALIZE IDLE STACK VALUES!
00D0 5F00  CALL CREATE_STACK(R0: ARGUMENT PTR
00D2 0000*  R1: TOP OF STACK
00D4 010F  ADD R15, #8 !OVERLAY ARGUMENTS!
00D6 0008  !

! ALLOCATE MMU IMAGE FOR IDLE PROCESS!
00D8 5F00  CALL ALLOCATE_MMU(R0: DBR # !
00DA 0000*  !

! PREPARE IDLE PROCESS MMU ENTRIES!
00DC 2101  LD R1, #STACK_SEG ! SEG #
00DE 0001  !
00E0 97F2  POP R2, @R15 IGET STACK ADDRESS
00E2 2103  LD R3, #0 I WRITE ATTRIBUTE!
00E4 0000  !
00E6 2104  LD R4, #STACK_BLOCK-1 I BLOCK LIMITS!
00E8 0000  !

! SAVE DBR #!
00EA 93F0  PUSH @R15, R0

! CREATE MMU IMAGE ENTRY!
00EC 5F00  CALL UPDATE_MMU_IMAGE(R1: SEGMENT #
00EE 0000*  R2: SEG ADDRESS
00F0 0000  R3: SEG ATTRIBUTES

- 323 -
R4: SEG LIMITS

I RESTORE DBR #1
POP R0, @R15

I GET MMU ADDRESS
CALL GET_DBR_ADDR (R0: DBR #)

I PREPARE VPT ENTRIES FOR IDLE PROCESS
LD R2, #0           I PRIORITY
LD R5, #OFF         I PREEMPT
LD R6, #OFF         I KERNEL PROC

I CREATE VPT ENTRIES
CALL UPDATEVP_TABLE (R1: DBR
R2: PRIORITY
R4: IDLE_FLAG
R5: PREEMPT
R6: EXT_VP_FLAG)
RETURNS:
R9: VP_ID

I ENTER VP ID OF IDLE PROCESS IN PRDS
LD PRDS.IDLE_VP, R9

I INITIALIZE IDLE VP'S
LD R2, #1           I PRIORITY
LD R5, #ON          I PREEMPT
LD R6, #ON          I NON-KERNEL PROC

I INITIALIZE VP VALUES
DO
CALL UPDATEVP_TABLE (R1: DBR
R2: PRIORITY
R4: IDLE_FLAG
R5: PREEMPT
R6: EXT_VP_FLAG)
RETURNS:
R9: VP_ID

DEC R0, #1
CP R0, #0
IF EQ ALL VP'S INITIALIZED THEN
EXIT

- 324 -
! INITIALIZE VPT HEADER !
! GET LOGICAL CPU NUMBER !
012A 6102  LD R2, PRDS.LOG_CPU_ID
0130 0002*  
0132 4D05  LD VPT.LOCK, #OFF
0134 0000*  
0136 0000  
0138 4D25  LD VPT.RUNNING_LIST(R2), #NIL
013A 0002*  
013C FFFF  
013E 4D25  LD VPT.READY_LIST(R2), #NIL
0140 0006*  
0142 FFFF  
0144 4D08  CLR VPT.FREE_LIST !HEAOD OF MSG LIST!
0146 000A*  

! THREAD VP'S BY PRIORITY AND SET STATES TO READY !
0148 8D28  CLR R2 !START WITH VP #1!

THREAD:
DO
014A 610D  LD R13, PRDS.LOG_CPU_ID
014C 0002*  
014E 7603  LDA R3, VPT.READY_LIST(R13)
0150 0006*  
0152 7604  LDA R4, VPT.VP.NEXT_READY_VP
0154 001C*  
0156 7605  LDA R5, VPT.VP.PRI
0158 0012*  
015A 7606  LDA R6, VPT.VP.STATE
015C 0014*  
015E 2107  LD R7, #READY
0160 0001  

! SAVE OBJ ID !
0162 93F2  PUSH @R15, R2
0164 5F00  CALL LIST_INSERT !R2: OBJ ID
0166 0000*  

R3: LIST_HEAD_PTR ADDR
R4: NEXT_OBJ_PTR
R5: PRIORITY_PTR
R6: STATE_PTR
R7: STATE 

! RESTORE OBJ ID !
0168 97F2  POP R2, @R15
016A 0102  ADD R2, #SIZEOF VP_TABLE
016C 0020  
016E 0B02  CP R2, #(NR_VP * (SIZEOF VP_TABLE))
0170 0100  
0172 5ED8  IF EQ THEN EXIT FROM THREAD FI
0174 017A'  

- 325 -
I INITIALIZE VP MESSAGE LIST
I NOTE: ONLY THE THREAD FOR THE MESSAGE LIST NEED BE CREATED AS ALL MESSAGES ARE INITIALLY AVAILABLE FOR USE. THE INITIAL MESSAGE VALUES WERE CREATED FOR CLARITY ONLY TO SHOW THAT THE MESSAGES HAVE NO USABLE INITIAL VALUE!

017C 8D18 CLR R1

MSG_LST_INIT:
I NOTE: R1 REPRESENTS CURRENT ENTRY IN MSG_LIST, R2 REPRESENTS CURRENT POSITION IN MSG_LIST ENTRY, AND R3 REPRESENTS NEXT ENTRY IN MSG_LIST.

DO
017E A112 LD R2, R1
0180 A123 LD R3, R2
0182 0103 ADD R3, #SIZEOF MESSAGE

FILL_MSG:
DO
0186 4D25 LD VPT.MSG_Q.MSG(R2), #INVALID
0188 0110* INC R2, #2
018A EE22 CP R2, R3
0190 5E08 IF EQ THEN EXIT FROM FILL_MSG FI
0192 0198* 0194 5E08
0196 019A* 0198 E8F6 OD
019A 4D15 LD VPT.MSG_Q.SENDER(R1), #NIL
019C 0120* 019E FFFF
01A0 A112 LD R2, R1
01A2 0103 ADD R1, #SIZEOF MSG_TABLE
01A4 0020 CP R1, #SIZEOF MSG_TABLE*NR VP
01A6 0B01 IF EQ THEN
01A8 0100 01AA 5E0E 01AC 018C* 01AE 4D25 LD VPT.MSG_Q.NEXT_MSG(R2), #NIL
01B0 0122* 01B2 FFFF 01B4 5E08 01B6 01C2* EXIT FROM MSG_LST_INIT
01B8 5E08 01BA 01C0* ELSE
01BC 6F21 LD VPT.MSG_Q.NEXT_MSG(R2), R1

- 326 -
I GET LOGICAL CPU # FOR USE
BY ITC GETWORK.

LD   R13, PRDS.LOG_CPU_ID

I BOOTSTRAP COMPLETE!
I START SYSTEM EXECUTION AT PREEMPT ENTRY!
I POINT IN ITC GETWORK PROCEDURE!

JP    BOOTSTRAP_ENTRY

END BOOTSTRAP
UPDATE VP TABLE
PROCEDURE

* INITIALIZES VPT ENTRIES *

** REGISTER USE: **
** PARAMETERs:**
* R1: DBR ADDRESS *
* R2: PRIORITY *
* R5: PREEMPT FLAG *
* R6: EXTERNAL VP FLAG *
** RETURNS:**
* R9: ASSIGNED VP ID *

** LOCAL VARIABLES:**
* R7: LOGICAL CPU # *
* R8: EXT_vp_LIST OFFSET *
* R9: VPT OFFSET *

ENTRY
GET OFFSET IN VPT FOR NEXT ENTRY !

LD R9, NEXT_AVAIL_VP
LD VPT.VP.DBR(R9), R1
LD VPT.VP.PRI(R9), R2
LD VPT.VP.IDLE_FLAG(R9), R6
LD VPT.VP.PREEMPT(R9), R5
LD R7, PHS.SLOG_CPU_ID
LD VPT.VP.PHYS_PROCESSOR(R9), R7
LD VPT.VP.NEXT_READY_VP(R9), #NIL
LD VPT.VP.MSG_LIST(R9), #NIL

! CHECK EXTERNAL VP FLAG !
CP R6, #ON
IF EQ !EXTERNAL VP!
THEN ! VP IS TC VISIBLE !

LD R8, NEXT_EXT_VP
LD EXT_vp_LIST(R8), R9
LD VPT.VP.EXT_ID(R9), R8

- 328 -
0206 A981     INC R8, #2
0208 6F08     LD NEXT_EXT_VP, R8
020A 0002*    
020C 5E08     ELSE !VP DOUND TO KERNEL PROCESS!
020E 0216*    
0210 4D05     LD VPT.VP.EXT_ID, #EIL
0212 0020*    
0214 FFFF     FI
0216 A19A     LD R10, R9
0218 010A     ADD R10, #SIZEOF VP_TABLE
021A 0020    
021C 6F0A     LD NEXT_AVAIL_VP, R10
021E 0000*    
0220 9E08     RET
0222          END UPDATE_VP_TABLE
              END BOOTSTRAP_LOADER
Appendix P

LIBRARY FUNCTION LISTINGS

Z8000ASM 2.02

LOC  OBJ CODE   STMT SOURCE STATEMENT

LIBRARY FUNCTION  MODULE

$LISTON $TTY

CONSTANT

KERNEL_PCW        := %5000
STACK_SEG_SIZE    := %100
STACK_BASE        := STACK_SEG_SIZE-%10
STATUS_REG_BLOCK  := STACK_SEG_SIZE-%10
INTERRUPT_FRAME   := STACK_BASE-4
INTERRUPT_REG     := INTERRUPT_FRAME-34
M_S_P             := INTERRUPT_REG-2
NIL                := %FFFF
LIST_INSERT PROCEDURE

* INSERTS OBJECTS INTO A LIST *
* BY ORDER OF PRIORITY AND SETS *
* ITS STATE *

PARAMETERS:
* R2: OBJECT ID *
* R3: HEAD_OF_LIST_PTR ADDR *
* R4: NEXT_OBJ_PTR ADDR *
* R5: PRIORITY_PTR ADDR *
* R6: STATE_PTR ADDR *
* R7: OBJECT STATE *

LOCAL VARIABLES:
* R8: HEAD_OF_LIST_PTR *
* R9: NEXT_OBJ_PTR *
* R10: CURRENT_OBJ_PRIORITY *
* R11: NEXT_OBJ_PRIORITY *

ENTRY
! GET FIRST OBJECT IN LIST !
0000 2138  LD    R8, R3
0002 0B08  CP    R8, #NIL
0004 FFFF
0006 5E0E  IF EQ !LIST IS EMPTY! THEN
0008 0018'
000A 2F32  LD    @R3, R2
000C 7449  LDA   R9, R4(R2)
000E 0200
0010 0D95  LD    @R9, #NIL
0012 FFFF
0014 5E0E  ELSE
0016 005A'
0018 715A  LD    R10, R5(R2) !OBJ PRI!
001A 0200
001C 715B  LD    R11, R5(R8) !HEAD PRI!
001E 0800
0020 8BBA  CP    R10, R11
0022 5E02  IF GT !OBJ PRI>HEAD PRI! THEN
0024 0030'
0026 2F32  LD    @R3, R2 !PUT AT FRONT!
0028 7348  LD    R4(R2), R8
002A 0200
002C 5E08  ELSE ! INSERT IN BODY OF LIST !
002E 005A'

SEARCH_LIST:
DO
CP R8, #NIL
IF EQ IEND OF LIST! THEN
EXIT FROM SEARCH_LIST
FI
LD R11, R5(R8) IGET NEXT PRI!
CP R10, R11
IF GT !CURRENT PRI>NEXT PRI! THEN
EXIT FROM SEARCH_LIST
FI
! GET NEXT OBJ!
LD R9, R8
LD R8, R4(R9)
OD I END SEARCH_LIST !
! INSERT IN LIST!
LD R4(R2), R8
LD R4(R9), R2
FI
FI
! SET OBJECT'S STATE!
LD R6(R2), R7
RET
END LIST_INSERT
CREATE_STACK  PROCEDURE

* INITIALIZES KERNEL STACK   *
* SEGMENT FOR PROCESSES       *

* REGISTER USE:           *
* PARAMETERS:            *
* R0: ARGUMENT POINTER   *
* (INCLUDES: FCW, IC, NSP, AND   *
* RETURN POINT. SEE LOCAL   *
* VARIABLES BELOW.)       *
* R1: TOP OF STACK        *
* R2-R14: INITIAL REGISTER *
* STATES. (NOTE: IN DEMO, NO   *
* SPECIFIC INITIAL REGISTER   *
* VALUES ARE SET, EXCEPT R13   *
* (USER ID) FOR USER PRO- *
* CESS.)                     *

* LOCAL VARIABLES         *
* (FROM ARGUMENTS STORED ON   *
* STACK.)                   *
* R3: FCW  *
* R4: PROCESS ENTRY POINT(IC) *
* R5: NSP   *
* R6: PREEMPT RETURN POINT   *

ENTRY

0060  93F0   PUSH  @R15, R0  ISAVE ARGUMENT PTR1
0062  ADF0   EI     R0, R15  ISAVE SP1
0064  341F   LDA   R15, R1 (#INTERUPT_REG)
0066  00CA   LDM   @R15, R1, #16  INITIAL REG. VALUES!
0068  1CF9   LDA   R15, R0  ILOAD ARGUMENTS!
006A  010F

! NOTE: ONLY REGISTERS R2-R14 MAY CONTAIN
INITIALIZATION VALUES !

006C  A10F   LD     R15, R0  IRESTORE SP1
006E  97F0   POP    R0, @R15 IRESTORE ARGUMENT PTR1
0070  A1FE   LD     R14, R15  ISAVE CALLER RETURN POINT1
0072  A10F   LD     R15, R0  IGET ARGUMENT PTR1
0074  1CF1   LDM   R3, @R15, #4  ILOAD ARGUMENTS1
0076  0303
0078  341F
007A  00EC
007C  1CF9   LDM   @R15, R3, #2  INIT IRET FRAME!
007E  0301
0080  341F   LDA   R15, R1 (#INTERUPT_FRAME)
0082  00C8
0084  2FF5   LD     @R15, R5  ISET NSP1
0086  030F   SUB    R15, #2
0088  0002
008A  2FF6   LD     @R15, R6  IPREEMPT RET POINT1
008C  3418   LDA   R8, R1 (#STACK_BASE)

- 333 -
008E 00F0  INITIALIZE STATUS REGISTER BLOCK
0090 2100  LD     R0, #KERNEL_FCW
0092 5000  
0094 1C89  LDM    @R8, R15, #2 !SAVE SP & FCW!
0096 0F01  
0098 A1EF  LD     R15, R14 !RESTORE RETURN POINT!
009A 9E08  RET
009C  END CREATE_STACK
       END LIBRARY_FUNCTION
Appendix G

INNER TRAFFIC CONTROLLER LISTINGS

Z8000ASH 2.02
LOC OBJ CODE STMT SOURCE STATEMENT

INNER_TRAFFIC_CONTROL MODULE

$LISTON $TTY

1. GETWORK:

A. NORMAL ENTRY DOES NOT SAVE REGISTERS.
   (THIS IS A FUNCTION OF THE GATEKEEPER).
B. 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.
C. THE PREEMPT INTERRUPT ENTRY HANDLER 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 THE MONITOR WITH AN ERROR CODE IN R0
   AND THE PC VALUE IN R1.
B. ITC PROCEDURES CALLING GETWORK PASS DBR
   (REGISTER R14) AND LOGICAL CPU NUMBER
   (REGISTER R13) AS INPUT PARAMETERS.
   (INCLUDES: SIGNAL, WAIT, SWAP_VDBR,
   PHYS_PREEMPT_HANDLER, AND IDLE).

CONSTANT

<table>
<thead>
<tr>
<th>CODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>U_L</td>
<td>Unauthorized lock</td>
</tr>
<tr>
<td>M_L_EM</td>
<td>Message list empty</td>
</tr>
<tr>
<td>M_L_ER</td>
<td>Message list error</td>
</tr>
<tr>
<td>B_L_E</td>
<td>Ready list empty</td>
</tr>
<tr>
<td>M_L_O</td>
<td>Message list overflow</td>
</tr>
<tr>
<td>S_W_A</td>
<td>Swap not allowed</td>
</tr>
<tr>
<td>V_L_E</td>
<td>VP index error</td>
</tr>
<tr>
<td>M_U</td>
<td>MMU unavailable</td>
</tr>
</tbody>
</table>
I ******** SYSTEM PARAMETERS ********!

NR_SDR  := 64  I LONG WORDS!
NR_CPU  := 2
NR_VP   := NR_CPU*4
NR_AVAIL_VP := NR_CPU*2
MAX_DBG_NR := 10  I PER CPU!
STACK_SEG  := 1
SRDS_SEG  := 0
STACK_SEG_SIZE := %100

I ***** OFFSETS IN STACK SEG *****!

STACK_BASE  := STACK_SEG_SIZE-%10
STATUS_REG_BLOCK := STACK_SEG_SIZE-%10
INTERRUPT_FRAME := STACK_BASE-4
INTERRUPT_REG := INTERRUPT_FRAME-34
N_S_P   := INTERRUPT_REG-2
P_C_W   := STACK_SEG_SIZE-%E

ON       := %FFFF
OFF      := 0
RUNNING  := 0
READY    := 1
WAITING  := 2
NIL      := %FFFF
INVALID  := %EEE
MONITOR  := %A900  I HBUG ENTRY I
KERNEL_FCW := %5000
AVAILABLE := 0
ALLOCATED := %FF

TYPE
MESSAGE ARRAY [16 BYTE]
ADDRESS WORD
VP_INDEX INTEGER
MSG_INDEX INTEGER

SEG_DESC_REG RECORD
[
  BASE ADDRESS
  ATTRIBUTES BYTE
  LIMITS BYTE
]

MMU          ARRAY[NR_SDR SEG_DESC_REG]

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

- 336 -
VP_TABLE RECORD
[ DBR ADDRESS
  PRI WORD
  STATE WORD
  IDLE_FLAG WORD
  PREEMPT WORD
  PHYS_PROCESSOR WORD
  NEXT READY VP VP_INDEX
  MSG_LIST MSG_INDEX
  EXT_ID WORD
  FILLER_1 ARRAY [ 7, WORD ]
]

EXTERNAL
LIST_INSERT PROCEDURE

GLOBAL
BOOTSTRAP_ENTRY LABEL

$SECTION ITC_DATA

0000 WPT RECORD
[ LOCK WORD
  RUNNING_LIST ARRAY [ NR_CPU WORD ]
  READY_LIST ARRAY [ NR_CPU WORD ]
  FREE_LIST MSG_INDEX
  VIRT_INT_VEC ARRAY [ 1, ADDRESS ]
  FILLER_2 WORD
  VP ARRAY [ NR_VP, VP_TABLE ]
  MSG_Q ARRAY [ NR_VP, MSG_TABLE ]
]

0210 EXT VP LIST ARRAY [ NR_AVAIL VP WORD ]

$SECTION MMU_DATA

0000 MMU_IMAGE RECORD
[ MMU_STRUCTURE ARRAY [ MAX DBR WR MMU ]
]

0A00 NEXT_AVAIL MMU ARRAY [ MAX DBR WR BYTE ]

0A0A PRDS RECORD
[ PHYS_CPU_ID WORD
  LOG_CPU_ID INTEGER
  VP_WR WORD
  IDLE VP VP_INDEX ]

- 337 -
$SECTION ITC_INT_PROC
INTERNAL
GETWORK PROCEDURE
*****************************************
* SWAPS VIRTUAL PROCESSORS *
* ON PHYSICAL PROCESSOR. *
*****************************************
* PARAMETERS:
* R13: LOGICAL CPU # *
* REGISTER USE: *
* STATUS REGISTERS *
* R14: DBR (SIMULATION) *
* R15: STACK_POINTER *
* LOCAL VARIABLES:
* R1: READY_VP (NEW) *
* R2: CURRENT_VP (OLD) *
* R3: FLAG CONTROL WORD *
* R4: STACK_SEG_BASE_ADDR *
* R5: STATUS_REG_BLOCK_ADDR *
* R6: NORMAL_STACK_POINTER *
*****************************************
ENTRY
GET_STACK_BASE!
ENTRY

0000 31E4  LD    R4, R14(#STACK_SEG*4)
0002 0004  LDA    R5, R4(#STATUS_REG_BLOCK)
0004 3445   
0006 00F0   ! ** SAVE SP ** !
0008 2F5F   LD    @R5, R15
000A 7D32   ! ** SAVE FCW ** !
000C 3343   LD    R4(#F_C_W), R3
000E 00F2   

BOOTSTRAP_ENTRY: ! GLOBAL LABEL !
GET_READY_VP_LIST!
ENTRY

0010 61D1  LD    R1, VPT_READY_LIST(R13)
0012 0006   

SELECT_VP:
DO ! UNTIL ELIGIBLE READY_VP FOUND !
ENTRY

0014 4D11  CP    VPT_VF.IDLE_FLAG(R1), #ON
0016 0016   IF EQ ! VP IS IDLE ! THEN
0018 5E0E   CP    VPT_VF.PREEMPT(R1), #ON
001A 0030   IF EQ ! PREEMPT INTERRUPT IS ON ! THEN
001C 4D11   CP    VPT_VF.IDLE_FLAG(R1), #ON
001E 0018   IF EQ ! VP IS IDLE ! THEN
0020 5E0E   CP    VPT_VF.PREEMPT(R1), #ON
0022 0030   IF EQ ! PREEMPT INTERRUPT IS ON ! THEN
0024 5E0E   CP    VPT_VF.IDLE_FLAG(R1), #ON
0026 002C   IF EQ ! VP IS IDLE ! THEN
0028 5E08   CP    VPT_VF.PREEMPT(R1), #ON
002A 003C   IF EQ ! PREEMPT INTERRUPT IS ON ! THEN

- 338 -
F
002C 5E08 ELSE I VP NOT IDLE !
002E 0034'
0030 5E08 EXIT FROM SELECT_VP
0032 003C'
F
! GET NEXT READY_VP !
0034 6113 LD R3, VPT.VP.NEXT_READY_VP(R1)
0036 001C'
0038 A131 LD R1, R3
003A E8EC 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
IDLING OR IF THERE ARE NO OTHER VP'S ON
THE READY_LIST. ONCE SCHEDULED, IT
WILL RUN UNTIL RECEIVING A HDWE INTERRUPT. !

! NOTE: R14 IS USED AS DBR HERE. WHEN MMU
IS AVAILABLE THIS SERIES OF SAVE AND LOAD
INSTRUCTIONS WILL BE REPLACED BY SPECIAL I/O
INSTRUCTIONS TO THE MMU. !

! PLACE NEW_VP IN RUNNING STATE !
003C 4D15 LD VPT.VP.STATE(R1), #RUNNING
003E 0014'
0040 0000
0042 6FD1 LD VPT.RUNNING_LIST(R13), R1
0044 0002'

! ** SWAP DBR ** !
0046 611E LD R14, VPT.VP.DBR(R1)
0048 0010'

! LOAD NEW_VP SP !
004A 31E4 LD R4, R14(#STACK_SEG*4)
004C 0004
004E 3445 LDA R5, R4(#STATUS_REG_BLOCK)
0050 00F0
0052 215F LD R15, @R5

! ** LOAD NEW FCW ** !
0054 3143 LD R3, R4(#F_C_W)
0056 00F2
0058 7D3A LDCTL FCW, R3
005A 9208 RET
005C END GETWORK
005C ENTER_MSG_LIST PROCEDURE

* INSERTS POINTER TO MESSAGE *
* FROM CURRENT_VP TO SIGMALED_VP *
* IN FIFO MSG_LIST *

* REGISTER USE: *
* PARAMETERS: *
* R8(R9): MSG (INPUT) *
* R1: SIGMALED_VP (INPUT) *
* R13: LOGICAL CPU NUMBER *
* LOCAL VARIABLES: *
* R2: CURRENT_VP *
* R3: FIRST_FREE_MSG *
* R4: NEXT_FREE_MSG *
* R5: NEXT_Q_MSG *
* R6: PRESENT_Q_MSG *

ENTRY

005C 61D2 LD R2, VPT.RUNNING_LIST(R13)
005E 0002'

I GET FIRST_MSG FROM FREE_LIST !

0060 6103 LD R3, VPT.FREE_LIST
0062 000A'

I * * * * DEBUG * * * * *
CP R3, #NIL
0064 0B03
0066 FFFF
0068 520E
006A 0078'
006C 7601 LDA R1, $
006E 006C'
0070 2100 LD R0, #M_L_01 MESSAGE LIST OVERFLOW !
0072 0004
0074 5F00 CALL MONITOR
0076 A900

PI *
* END DEBUG * * *

0078 6134 LD R4, VPT.MSG_Q.NEXT_MSG(R3)
007A 0122'
007C 6F04 LD VPT.FREE_LIST, R4
007E 000A'

I INSERT MESSAGE LIST INFORMATION !

0080 763A LDA R10, VPT.MSG_Q.MSG(R3)
0082 0110'
0084 2107 LD R7, #SIZEOF MESSAGE
0086 0010
0088 BA81 LDIRB @R10, @R8, R7
008A 07A0
008C 6F32 LD VPT.MSG_Q.SENDER(R3), R2
008E 0120'

- 340 -
I INSERT MSG IN MSG_LIST!

LD R5, VPT.VP.MSG_LIST(R1)
0092 001E'

CP R5, #NIL
0094 0B05

IF EQ ! MSG LIST IS EMPTY! THEN
0096 FFFF
0098 5E0E
009A 00A4'

! INSERT MSG AT TOP OF LIST!
LD VPT.VP.MSG_LIST(R1), R3
009E 001E'

ELSE! INSERT MSG IN LIST!
00A0 5E08

MSG_Q_SEARCH:
DO ! WHILE NOT END OF LIST!
00A2 00BC'

CP R5, #NIL
00A4 0B05

IF EQ ! END OF LIST! THEN
00A6 FFFF
00A8 5E0E
00AA 00B0'

EXIT FROM MSG_Q_SEARCH
00AC 5E08
00AE 00B8'

FI

! GET NEXT LINK!
00B0 A156
LD R6, R5
00B2 6165
LD R5, VPT.MSG_Q.NEXT_MSG(R6)
00B4 0122'

OD
! INSERT MSG IN LIST!
00B6 88F6

LD VPT.MSG_Q.NEXT_MSG(R6), R3
00B8 6F63
00BA 0122'

FI
00BC 6F35
LD VPT.MSG_Q.NEXT_MSG(R3), R5
00BE 0122'

RET
00C0 9E08

END ENTER_MSG_LIST

- 341 -
GET_FIRST_MSG

*************PROCEDURE
! *************
* REMOVES MSG FROM MSG_LIST
* AND PLACES ON FREE LIST.
* RETURNS SENDER'S MSG AND
* VP_ID
*************

*REGISTER USE:
* PARAMETERS:
* R8(R9): MSG POINTER (INPUT)
* R13: LOGICAL CPU NUMBER (INPUT)
* R1: SENDER VP (RETURNED)
* LOCAL VARIABLES
* R2: CURRENT_VP
* R3: FIRST_MSG
* R4: NEXT_MSG
* R5: NEXT_FREE_MSG
* R6: PRESENT_FREE_MSG

ENTRY

00C2 61D2
00C4 0002* 

LD R2, VPT.RUNNING_LIST(R13)

00C2 61D2
00C6 6123
00C8 001E* 

LD R3, VPT.VP.MSGJ.IST(R2)

00C6 6123
00CA 0B03
00CC FFF7
00CE 5F0E* 

CP R3, $NIL

00D0 00DE* 

IF EQ THEN

00C6 6123
00D2 2100
00D4 0001
00D6 7601
00D8 00D6* 

LD R0, #M_L_EM & MSG LIST EMPTY!

00D8 00D6* 

LDA R1, $

00DA 5F00
00DC A900

CALL MONITOR

00DE 6134
00E0 0122* 

LD R4, VPT.MSG_Q.NEXT_MSG(R3)

00E2 6F24
00E4 001E* 

LD VPT.VP.MSG_LIST(R2), R4

00EE 6123
00E6 6105
00E8 000A* 

CP R5, $NIL

00EA 0B05
00EC FFFF
00EE 520E
00F0 0100* 

IF EQ ! FREE_LIST IS EMPTY! THEN

00F2 6F03
00F4 000A* 

LD VPT.FREE_LIST, R3

! INSERT AT TOP OF LIST!

- 342 -
00F6 4D35    LD    VPT.MSG_Q.NEXT_MSG(R3), #NIL
00F8 0122'  ELSE I INSERT IN LIST !
00FA FFFF    FREE_Q_SEARCH:
00FC 5F08
00FE 011C'
               DO
0100 0B05    CP    R5, #NIL
0102 FFFF    IF EQ I END OF LIST ! THEN
0104 5B0E    EXIT FROM FREE_Q_SEARCH
0106 010C'    FI
               I GET NEXT MSG !
0108 5F08
010A 0114'
               OD
010C A156    LD    R6, R5
010E 6165    LD    R5, VPT.MSG_Q.NEXT_MSG(R6)
0110 0122'    I INSERT IN LIST !
0112 E8F6    OD
0114 6F63    LD    VPT.MSG_Q.NEXT_MSG(R6), R3
0116 0122'    I GET MESSAGE INFORMATION:
               (RETURNS R1: SENDING_VP) !
0118 6F35    LD    VPT.MSG_Q.NEXT_MSG(R3), R5
011A 0122'    FI
               I GET MESSAGE INFORMATION:
011C 6131    (RETURNS R1: SENDING_VP) !
011E 0120'    LD    R1, VPT.MSG_Q.SENDER(R3)
0120 763A    LDA    R10, VPT.MSG_Q.MSG(R3)
0122 0110'    OD
0124 2107    LD    R7, #SIZEOF MESSAGE
0126 0010    LDIRB    @R9, @R10, R7
0128 BAA1
012A 0780
012C 9E08    RET
012E            END GET_FIRST_MSG

- 343 -
I * * INNER TRAFFIC CONTROL ENTRY POINTS * * !

I NOTE: ALL INTERRUPTS MUST BE MASKED WHENEVER THE VPT IS LOCKED. THIS IS TO PREVENT AN EMBRACE FROM OCCURRING SHOULD AN INTERRUPT OCCUR WHILE THE VPT IS LOCKED. !

GLOBAL
$SECTION ITC_GLB_PROC

PREEMPT_RET LABEL
KERNEL_EXIT LABEL
CREATE_INT_VEC PROCEDURE

* * * *************** ***************
* * CREATES ENTRY IN VIRTUAL INT-*
* * EROPT VECTOR WITH ADDRESS *
* * OF THE VIRTUAL INTERRUPT HAN-*
* * DLER. *
* * *************** ***************
* * PARAMETERS: *
* * R1: VIRTUAL INTERRUPT # *
* * R2: INTERRUPT HANDLER ADDR *
* * *************** ***************

ENTRY

! COMPUET OFFSET IN VIRTUAL
INTERRUPT VECTOR !

0000 1900 MULT BRO, #SIZEOF ADDRESS
0002 0002 ! SAVE ADDRESS OF VIRTUAL INTERRUPT
HANDLER IN INTERRUPT VECTOR !
0004 6F12 LD VPT.VIRT_INT_VEC(R1), R2
0006 000C 9E08 RET
0008 9E08 000A END CREATE_INT_VEC
GET_DBR_ADDR

PROCEDURE

* CALCULATES DBR ADDRESS FROM *
* DBR NUMBER *

* REGISTER USE: *
* PARAMETERS: *
* RO: DBR # *

* RETURNS: *
* R1: DBR ADDRESS *

ENTRY

! GET BASE ADDRESS OF MNU IMAGE !

000A 7601 LDA R1, MNU_IMAGE
000C 0000

! ADD DBR HANDLE (OFFSET) TO MNU BASE ADDRESS TO OBTAIN DBR ADDRESS !

000E 8101 ADD R1, R0
0010 9E08 RET
0012 END GET_DBR_ADDR
ALLOCATE_HNU PROCEDURE

* ALLOCATES NEXT AVAILABLE MNU *
* IMAGE AND CREATES PRDS ENTRY *

* REGISTER USE: *
* RETURNS: *
* RO: DBR # *
* LOCAL VARIABLES: *
* R1: SEGMENT # *
* R2: PRDS ADDRESS *
* R3: PRDS ATTRIBUTES *
* R4: PRDS LIMITS *

ENTRY

! NOTE: THE FOLLOWING IS A SAFE SEQUENCE AS NEXT_AVAL_HNU AND HNU ARE CPU LOCAL!

GET_DBR:

DO

CPB NEXT_AVAL_HNU(R1), #AVAILABLE

IF EQ !HNU ENTRY IS AVAILABLE!

THEN

LDB NEXT_AVAL_HNU(R1), #ALLOCATED

EXIT FROM GET_DBR

ELSE !CURRENT ENTRY IS ALLOCATED!

INC R1, #1

ADD RO, #SIZEOF HNU

I *** DEBUG *** !

CP R1, #MAX_DBR_NR

IF EQ THEN

LD RO, #H_U IMMU UNAVAILABLE!

LDA R1, $

CALL MONITOR

FI

I *** END DEBUG *** !

FI
004A 2101  LD   R1, #PRDS_SEG  I SEGMENT NO. !
004C 0000  
004E 7602  LDA  R2, PRDS  I PRDS ADDR !
0050 0A0A'  
0052 2103  LD   R3, #1 ! READ ATTR !
0054 0001  
0056 2104  LD   R4, #((SIZEOF PRDS)-1)/256
0058 0000  ! PRDS LIMITS !

! CREATE PRDS ENTRY IN MMU IMAGE !
005A 5F00  CALL  UPDATE_MMU_IMAGE I(R1: SEGMENT #
005C 0060'   R2: SEG ADDRESS
            R3: ATTRIBUTES
            R4: SEG LIMITS)!

005E 9E08  RET
0060       END ALLOCATE_MMU
UPDATE_MNU_IMAGE  PROCEDURE

* CREATES SEGMENT DESCRIPTOR *
* ENTRY IN MNU IMAGE *

* REGISTER USE: *
* PARAMETERS: *
  * R0: DBR # *
  * R1: SEGMENT # *
  * R2: SEGMENT ADDRESS *
  * R3: SEGMENT ATTRIBUTES *
  * R4: SEGMENT LIMITS *
* LOCAL VARIABLES: *
  * R10: MNU BASE ADDRESS *
  * R13: OFFSET VARIABLE *

ENTRY

0060 210A  LD  R10, #MNU_IMAGE I MNU BASE ADDRESS I
0062 0000' ADD  R10, R0
0064 810A  LD  R13, #SIZEOF SEG_DESC_BEGIN
0066 0004  ADD  R10, R13 IADD OFFSET TO BASE ADDRESS!
0068 991C  MUL  RR12, R1 I COMPUTE SEG_DESC OFFSET I
006A 81DA  ADD  R10, R13 IADD OFFSET TO BASE ADDRESS!
006C 2FA2  LD  &R10, R2
0070 A9A1  INC  R10, #2
0072 0DA8  CLR  &R10
0074 2EAC  LDB  &R10, RL4
0076 A9A0  INC  R10, #1
0078 20AC  LDB  RL4, &R10
007A 0A0B  CPB  RL3, #%(2)00001000 I EXECUTE I
007C 0808  CPB  RL3, #%(2)00001000 I EXECUTE I
007E 5B02  IF  EQ THEN
0080 008A' ANDB  RL4, #%(2)11110111 I EXECUTE MASK I
0082 F7F7  ELSE
0086 5B08  ORB  RL4, RL3
0088 008E' ANDB  RL4, #%(2)11111110 I READ MASK I
008A 060C  ORB  RL4, RL3
008C FEFE  FI
008E 84BC  LDB  &R10, RL4
0090 2EAC  LDB  &R10, RL4
0092 9E08  RET

END UPDATE_MNU_IMAGE
WAIT PROCEDURE

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
* INTRA_KERNEL SYNC/COM PRIMATIVE *
* INVOKED BY KERNEL PROCESSES *
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
* PARAMETERS *
* R8 (R9): MSG POINTER (INPUT) *
* R1: SENDING_VP (RETURN) *
* GLOBAL VARIABLES *
* R14: DBR (PARAM TO GETWORK) *
* LOCAL VARIABLES *
* R2: CURRENT_VP (RUNNING) *
* R3: NEXT_READY_VP *
* R4: LOCK_ADDRESS *
* R13: LOGICAL CPU NUMBER *
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ENTRY

I MASK INTERRUPTS !
DI VI
I LOCK VPT !
LDA R4, VPT.LOCK
CALL SPIN_LOCK ! (R4: VPT.LOCK) !
NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
I GET CPU NUMBER !
CALL GET_CPU_NO !RETURNS:
               R1:CPU #
               R2: # VP'S
LD R13, R1
LD R2, VPT.RUNNING_LIST(R13)
LD R3, VPT.VP.NEXT_READY_VP(R2)
CP VPT.VP.MSG_LIST(R2), #NIL
IF EQ ! CURRENT VP'S MSG LIST IS EMPTY ! THEN
I REMOVE CURRENT_VP FROM READY_LIST !
I * * * * DEBUG * * * * !
CP R3, #NIL
IF EQ THEN
LD R0, #R_L_E ! READY LIST EMPTY !
LDA R1, $
CALL MONITOR
- 349 -
FI
! * * * END DEBUG * * * !

00CA 6FD3  LD  VPT.READY_LIST(R13), R3
00CE 0006'  
00CE 4D25  LD  VPT.VP.NEXT_READY_VP(R2), #NIL
00D0 001C'  
00D2 FFFF

! PUT IT IN WAITING STATE !
00D4 4D25  LD VPT.VP.STATE(R2), #WAITING
00D6 0014'  
00D8 0002

! SET DBR !
00DA 612E  LD  R14, VPT.VP.DBR(R2)
00DC 0010'

! SCHEDULE FIRST ELIGIBLE READY VP !
00DE 93F8  PUSH  @R15, R8
00E0 93F9  CALL  GETWORK  IR13:CPU #

! SCHEDULE FIRST ELIGIBLE READY VP !
00E2 5F00  CALL  GET_FIRST_MSG  IR13:CPU #
00E4 0000'

! RESTORE CPU # !
00E6 97FD  POP  R13, @R15
00E8 97F8  POP  R8, @R15

FI
! GET FIRST MSG ON CURRENT VP'S MSG LIST !
00EA 5F00  CALL  GET_FIRST_MSG  I COPIES MSG IN MSG ARRAY!
00EC 00C2'

! R13: LOGICAL CPU # !
!RETURNS R1:SENDER_VP !

! UNLOCK VPT !
00EE 4D08  CLR  VPT.LOCK
00F0 0000'

! UNMASK VECTORED INTERRUPTS !
00F2 7C05  EI  VI

! RETURN: R1:SENDER_VP !
00F4 9E08  RET
00F6  END WAIT

- 350 -
SIGNAL PROCEDURE

* INTRA_KERNEL SYNC /COM PRIMITIVE *
* INVOKED BY KERNEL PROCESSES *

* REGISTER USE: *
* PARAMETERS: *
* R8(R9): MSG POINTER (INPUT) *
* R1: SIGNALED VP_ID (INPUT) *

* GLOBAL VARIABLES *
* R13: CPU # (PARAM TO GETWORK) *
* R14: DBR (PARAM TO GETWORK) *

* LOCAL VARIABLES: *
* R1: SIGNALED VP *
* R2: CURRENT_VP *
* R4: VPT.LOCK ADDRESS *

ENTRY

1 SAVE VP ID 1
00F6 93F1 PUSH &R15, R1
1 MASK INTERRUPTS 1
00F8 7C01 DI VI
1 LOCK VPT 1
00FA 7604 LDA R4, VPT.LOCK
00FC 0000* 2
00FE 5F00 CALL SPIN_LOCK ! (R4:-VPT.LOCK) 1
0100 0282* 3

1 NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. 1
1 GET LOGICAL CPU # 1
0102 5F00 CALL GET_CPU_NO !RETURNS:
0104 02C8* 7
1 CPU #
R1:CPU #
R2:# VP'S!
0106 A11D LD R13, R1
1 RESTORE VP ID 1
0108 97F1 POP R1, &R15
1 PLACE MSG IN SIGNALED_VP'S MSG_LIST 1
010A 5F00 CALL ENTER_MSG_LIST !(R8:MSG_POINTER
R1:SIGNALED_VP
R13:LOGICAL CPU #) 1
010C 005C* 9
1 VPT.VP.STATE(R1), #WAITING
010E 4D11 CP R11:0014* 11
0110 0002 12
0112 5E0E IF EQ ! SIGNALED_VP IS WAITING ! THEN
0114 0148* 14

1 WAKE IT UP AND MAKE IT READY 1
0116 A112 LD R2, R1
011A 76D3 LDA R3, VPT.READY_LIST(R13)
011C 0066* 17
011E 7B 12 JA R4, VPT.VP:NEXT_READY_VP
0120 00 . .

- 351 -
LDA R5, VPT.VP.PRI
LDA R6, VPT.VP.STATE
LD R7, #READY

I SAVE LOGICAL CPU #1!

PUSH @R15, R13
CALL LIST_INSERT !R2: OBJ ID

R3: LIST_PTR_ADDR
R4: NEXT_OBJ_PTR
R5: PRIORITY_PTR
R6: STATE_PTR
R7: STATE

I RESTORE LOGICAL CPU #1!

POP R13, @R15
LD R2, VPT.RUNNING_LIST(R13)
LD VPT.VP.STATE(R2), #READY

I SET DBR!

LD R14, VPT.VP.DBR(R2)

I SCHEDULE FIRST ELIGIBLE READY VP!

CALL GETWORK !R13:LOGICAL CPU #

R14:DBR !

I UNLOCK VPT!
CLR VPT.LOCK

I UNMASK VECTORED INTERRUPTS!
EI VI

RET

END SIGNAL
SET_PRiEMPT  PROCEDURE

* SETS PRiEMPT INTERRUPT ON*
* TARGET_VP. CALLED BY TC_ *
* ADVANCE. *

*********************************************************************************
* REGISTER USE:
* PARAMETERS:
* R1:TARGET_VP_ID (INPUT) *
* LOCAL VARIABLES *
* R1: VP_INDEX *
*********************************************************************************

ENTRY

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

! CONVERT VP_ID TO VP_INDEX !
0150 6112   LD     R2, EXT_VP_LIST(R1)
0152 0210    

! TURN ON TGT_VP PREEMPT FLAG !
0154 4D25   LD     VPT.VP.PREEMPT(R2), #ON
0156 0018    
0158 PFFF    

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

015A 9E08   RET
015C        END SET_PRiEMPT
015C  IDLE  PROCEDURE
!**************************************************************************
* LOADS IDLE DBR ON   * 
* CURRENT VP. CALLED BY * 
* TC_GETWORK.         * 
**************************************************************************
* REGISTER USE         * 
* GLOBAL VARIABLE     * 
* R13: LOG CPU #      * 
* R14: DBR            * 
* LOCAL VARIABLES:    * 
* R2: CURRENT_VP     * 
* R3: TEMP VAR        * 
* R4: VPT.LOCK ADDR   * 
* R5: TEMP            * 
**************************************************************************
ENTRY
! GET LOGICAL CPU # !
CALL  GET_CPU_NO  !RETURNS:

  ! LOAD IDLE DBR ON CURRENT VP !
0174 6103  LD  R3, PRDS.IDLE_VP
0176 0A10'  
0178 6135  LD  R5, VPT.VP.DBR(R3)
017A 0010'  
017C 6F25  LD  VPT.VP.DBR(R2), R5
017E 0010'  

  ! TURN ON CURRENT VP'S IDLE FLAG !
0180 4D25  LD  VPT.VP.IDLE_FLAG(R2), #ON
0182 0016'  
0184 FFFF  ! SET VP TO READY STATE !
0186 4D25  LD  VPT.VP.STATE(R2), #READY
0188 0014'  
018A 0001  

  ! SCHEDULE FIRST ELIGIBLE READY VP !
018C 5F00  CALL  GETWORK  !R13:LOGICAL CPU #
018E 0000'  

  ! UNLOCK VPT !
0190 4D08  CLR  VPT.LOCK
0192 0000'  

  ! UNMASK VECTORED INTERRUPTS !
0194 7C05  EI  VI
0196 9E08  RET
0198   END IDLE

- 354 -
SWAP_VDBR
PROCEDURE

* LOADS NEW DBR ON *
* CURRENT VP. CALLED BY *
* TC_GETWORK. *

* REGISTER USE *
* PARAMETERS *
* R1: NEW_DBR (INPUT) *
* GLOBAL VARIABLES *
* R13: LOGICAL CPU # *
* R14: DBR *
* LOCAL VARIABLES *
* R2: CURRENT_VP *
* R4: VPT.LOCK_ADDR *

ENTRY

! SAVE NEW DBR !

0198 93F1 PUSH &R15, R1
! MASK INTERRUPTS !

019A 7C01 DI VI
! LOCK VPT !

019C 7604 LDA R4, VPT.LOCK

019E 0000' CALL SPIN_LOCK ! (R4:=VPT.LOCK) !

01A0 5F00 ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP.! I GET CPU # !

01A4 5F00 CALL GET_CPU_NO !RETURNS:

01A6 02C8' R1: CPU #
R2: # VP'S 51

01A8 A11D LD R13, R1
! GET CURRENT VP !

01AA 61D2 LD R2, VPT.RUNNING_LIST(R13)

01AC 0002' !** DEBUG **!

01AE 4D21 CP VPT.VP.MSG_LIST(R2), #NIL

01B0 001E' IF NE I MSG WAITING ! THEN

01B2 FFFF

01B4 5806 LD RO, #S_H_A ! SWAP NOT ALLOWED !

01B6 01C4' 01BA 0005
01BC 7601 LDA R1, $ IPCI

01BE 01BC' 01C0 5F00 CALL MONITOR

01C2 A900

01C4 612E 01C6 0010'

! SET DBR !

01C6 0010'

! RESTORE NEW DBR !
01C8 97F0   POP    R0, @R15
01CA 5F00   CALL   GET_DBR_ADDR 1 (R0: DBR #)
01CC 000A   

            RETURNS
            (R1: DBR_ADDR) !

01CE 6F21   LD      VPT.VP.DBR(R2), R1
01D0 0010   

            I LOAD NEW DBR ON CURRENT VP !

01D2 4D25   LD      VPT.VP.IDLE_FLAG(R2), #OFF
01D4 0016   01D6 0000

            I TURN OFF IDLE FLAG !

01D8 4D25   LD      VPT.VP.STATE(R2), #READY
01DA 0014   01DC 0001

            I SET VP TO READY STATE !

01DE 5F00   CALL   GETWORK 1R13:LOGICAL CPU #
01E0 0000   01E2 4D08

            I SCHEDULE FIRST ELIGIBLE READY VP !

01E4 0000   CLR    VPT.LOCK

            I UNLOCK VPT !

01E6 7C05   EI      VI

01E8 9E08   RET
01EA       END SWAP_VDBR

- 356 -
PHYS_PREEMPT_HANDLER  PROCEDURE

* HARDWARE PREEMPT INTERRUPT
* HANDLER. ALSO TESTS FOR
* VIRTUAL PREEMPT INTERRUPT
* FLAG AND INVOKES INTERRUPT
* HANDLER IF FLAG IS SET.
* INVOKED UPON EVERY EXIT FROM
* KERNEL. KERNEL FCW MASKS
* VPN INTERRUPTS TO PREVENT
* SIMULTANEOUS PREEMPT INTERRUPT.
* HANDLING.

REGISTER USE

LOCAL VARIABLES

R1: PREEMPT_INT_FLAG
R2: CURRENT_VP
GLOBAL VARIABLES

R13: LOGICAL CPU #
R14: DBR

ENTRY

** PREEMPT_HANDLER ** !

SAVE ALL REGISTERS !

SUB R15, #32
LDM @R15, R1, #16

SAVE NORMAL STACK POINTER (NSP) !

LDCTL R6, NSP
PUSH @R15, R6
GET CPU # 1
CALL GET_CPU_NO

RETURNS:

B1: CPU #
R2: # VP'S I

LD R13, R1
MASK INTERRUPTS I
DI VI
LOCK VPT I
LDA R4, VPT.LOCK
CALL SPIN_LOCK

RETURNS WHEN VPT IS LOCKED!
SET DBR I

LD R2, VPT.RUNNING_LIST(R13)
CALL SPIN_LOCK

LD R14, VPT.VP.DBR(R2)

- 357 -
I PUT CURRENT PROCESS IN READY STATE!

020E 4D25   LD   VPT.VP.STATE(R2), #READY
0210 0014*
0212 0001
0214 5F00   CALL   GETWORK  IR13:LOG CPU #
0216 0000*

R14: DBR !

PREEMPT_RET:
I UNLOCK VPT !
0218 4D08   CLR   VPT.LOCK
021A 0000*
I UNMASK VECTORED INTERRUPTS !

EI   VI

KERNEL_EXIT:
I *** UNMASK VIRTUAL PREEMPTS *** !
I ** NOTE: SAFE SEQUENCE AND DOES NOT REQUIRE
VPT TO BE LOCKED. ** !

I GET CURRENT_VP !

021E 610D   LD   R13, PRDS.LOG_CPU_ID
0220 00C0*
0222 61D2   LD   R2, VPT.RUNNING_LIST(R13)
0224 0002*

I TEST PREEMPT INTERRUPT FLAG !

0226 4D21   CP   VPT.VP.PREEMPT(R2), #0W
0228 0018*
022A FFFF
022C 5E08   IF   EQ I PREEMPT FLAG IS ON I THEN
022E 0240*

I RESET PREEMPT FLAG !

0230 4D25   LD   VPT.VP.PREEMPT(R2), #0F0F
0232 0018*
0234 0000
I SIMULATE VIRTUAL PREEMPT INTERRUPT !

0236 2101   LD   R1, #0
0238 0000
023A 6112   LD   R2, VPT.VIRT_INT_VEC(R1)
023C 000C*
023E 1E28   JP   #R2

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

I *** END VIRTUAL PREEMPT HANDLER *** !
FI

I NOTE: SINCE A HDWE INTERRUPT DOES NOT EXIT
THROUGH THE GATE, THOSE FUNCTIONS PROVIDED
BY A GATE EXIT TO HANDLE PREEMPTS MUST BE
PROVIDED HERE ALSO. !
I RESTORE NSP I
0240 97F6  POP R6, @R15
0242 7D6F  LDCTL NSP, R6
I RESTORE ALL REGISTERS I
0244 1CF1  LDH  R1, @R15, #16
0246 010F  ADD  R15, #32
0248 0020  
I EXECUTE HARDWARE INTERRUPT RETURN I
024C 7B00  IRET
024E      END PHYS_PREEMPT_HANDLER
**RUNNING_VP**

**PROCEDURE**

**CALL**

*Called by Traffic Control.*
*Returns VP_ID. Result is valid only while APT is locked.*

**Register Use**

*Parameters*
* R1: EXT_VP_ID (Returned) *
* R3: LOG_CPU # (Returned) *
* Local Variables *
* R2: VP INDEX *

**ENTRY**

* Mask Interrupts !

**DI**

* Lock VPT !

**LDA**

* R4, VPT.LOCK *

**CALL**

* SPIN_LOCK ! (R4:=VPT.LOCK) !

**ENTRY**

* NOTE: Returns when VPT is locked by this VP !

**CALL**

* GET_CPU_NO !returns: *

**ENTRY**

* R1: CPU # *

* R2: VP'S !

**LD**

* R3, R1 *

**LD**

* R2, VPT.RUNNING_LIST(R3) *

**ENTRY**

* Convert VP_INDEX to VP_ID *

**LD**

* R1, VPT.VP.EXT_ID(R2) *

**ENTRY**

* **DEBUG** **!**

**CP**

* R1, #NIL *

**IF EQ**

* KERNEL PROC ! THEN *

**LD**

* R0, #V_I_E ! VP INDEX ERROR !

**END DEBUG**

**ENTRY**

* Unlock VPT !

**CLR**

* VPT.LOCK *

**ENTRY**

* Unmask vectored interrupts !

**EI**

* VI *

**RET**

**END RUNNING_VP**

- 360 -
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 ** !

0282 0D41 CP @R4, $0FF
0284 0000
0286 5E06 IF NE ! NOT UNLOCKED ! THEN
0288 0296 TSET
028A 2100 LD R0, #U_L ! UNAUTHORIZED LOCK !
028C 0000
028E 7601 LDA R1, $
0290 029E JR R4
0292 5F00 CALL MONITOR
0294 A900 CALL TSET

FI
! ** END DEBUG ** !

TEST_LOCK:
! DO WHILE STRUCTURE LOCKED!

0296 0D46 TSET @R4
0298 E5FE JR MI, TEST_LOCK
! ** NOTE SEE PLZ/ASM MANUAL
FOR RESTRICTIONS ON
USE OF TSET. ** !

029A 9E08 RET

029C END SPIN_LOCK
029C

ITC_GET_SEG_PTR

procedure

* GETS BASE ADDRESS OF SEGMENT *
* INDICATED. *

* REGISTER USE: *
* R0: SEG BASE ADDRESS (RET) *
* R1: SEG NR (INPUT) *
* R2: RUNNING_VP (LOCAL) *
* R3: DBR_VALUE (LOCAL) *
* R4: VPT.LOCK *
* R13: LOGICAL CPU # *

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

entry

! SAVE SEGMENT # 1
029C 93F1 PUSH +R15, R1
! MASK INTERRUPTS!
029E 7C01 DI VI
! LOCK VPT!
02A0 7604 LDA R4, VPT.LOCK
02A2 0000' CALL SPIN_LOCK !R4:-VPT.LOCK!
02A4 5F00 CALL GET_CPU_NO !RETURNS:
02A6 0282' R1: CPU # 
02A8 5F00 CALL GET_CPU_NO !RETURNS:
02AA 02C8' R2: VP'S!
02AC A11D LD R13, R1
! RESTORE SEGMENT # 1
02AE 97F1 POP R1, R15
02B0 61D2 LD R2, VPT.RUNNING_LIST (R13)
02B2 0002' 02B4 6123 LD R3, VPT.VP.DBR (R2)
02B6 0010' ! UNLOCK VPT!
02B8 4D08 CLR VPT.LOCK
02BA 0000' ! UNMASK VECTORED INTERRUPTS!
02BC 7C05 EI VI
02BE 1900 MUL R0, $4
02C0 0004 02C2 7130 LD R0, R3 (R1)
02C4 0100
02C6 9E08 RET
02C8 END ITC_GET_SEG_PTR

- 362 -
GET_CPU_NO PROCEDURE

* FIND CURRENT CPU_NO *
* CALLED BY DIST &GR *
* AND MM *
* *****************************************
* RETURNS *
* R1: CPU_NO *
* R2: # OF VP'S *
* *****************************************

ENTRY
02C8 6101 LD R1, PRDS.LOG_CPU_ID
02CA 0A0C' LD R2, PRDS.VP_NR
02CC 6102 RET
02D0 9E08 END GET_CPU_NO

K_LOCK PROCEDURE

* STUB FOR WAIT LOCK *
* *****************************************
* R4: -LOCK (INPUT) *
* *****************************************

ENTRY
02D2 5F00 CALL SPIN_LOCK
02D4 0282' RET
02D6 9E08 END K_LOCK

K_UNLOCK PROCEDURE

* STUB FOR WAIT UNLOCK *
* *****************************************
* R4: -LOCK (INPUT) *
* *****************************************

ENTRY
02D8 0D48 CLR $R4
02DA 9E08 RET
02DC END K_UNLOCK

END INNER_TRAFFIC_CONTROL

- 363 -
Appendix H
SEGMENT MANAGER LISTINGS

Z8000ASM 2.02
LOC  OBJ  CODE  STMNT  SOURCE  STATEMENT

$LISTON $TTY
SEG_MGR       MODULE

CONSTANT
NULL_SEG := -1
NULL_ACCESS := 4
MAX_SEG_NO := 64
MAX_NO_KST_ENTRIES := 54
MAX_SEG_SIZE := 128
KST_SEG_NO := 2
NR_OF_KSEGS := 10
TRUE := 1
FALSE := 0
READ := 1
WRITE := 0

! *** SUCCESS_CODES *** !
SUCCEEDED := 2
SEGMENT_NOT_KNOWN := 22
ACCESS_CLASS_NOT_EQ := 33
NOT_COMPATIBLE := 24
SEGMENT_TOO_LARGE := 25
NO_SEG_AVAIL := 27
SEGMENT_NOT_KNOWN := 28
SEGMENT_IN_CORE := 29
KERNEL_SEGMENT := 30
INVALID_SEGMENT_NO := 31
NO_ACCESS_PERMITTED := 32
LEAF_SEG_EXISTS := 10
NO_LEAF_EXISTS := 11
ALIAS_DOES_NOT_EXIST := 23
NO_CHILD_TO_DELETE := 20
G_AST_FULL := 12
L_AST_FULL := 13
PROC_CLASS_NOT_GE_SEG_CLASS := 41
LOCAL_MEMORY_FULL := 16
GLOBAL_MEMORY_FULL := 17
SEC_STOR_FULL := 21

- 364 -
MONITOR := %059A

TYPE
H_ARRAY ARRAY [ 3 WORD ]

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

ADDRESS WORD
SEG_ARRAY ARRAY [ MAXSEG_SIZE BYTE ]

INTERNAL

$SECTION SM_KST_DCL
NOTE: THIS SECTION IS AN "OVERLAY"
OR "FRAME" USED TO DEFINE THE
FORMAT OF THE KST. NO STORAGE
IS ASSIGNED BUT RATHER THE KST IS
STORED IN A SEPARATELY OBTAINED
AREA (A SEGMENT SET ASIDE FOR IT) !

$ABS 0
0000 KST_ARRAY MAX_NO_KST_ENTRIES KST_REC

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
PROCESS_CLASS PROCEDURE
ITC_GET_SEG_PTR PROCEDURE
GET_DDR_NUMBER PROCEDURE

- 365 -
ADD R13,R9 !ADD OFFSET TO KST BASE ADDRESS!

LD R6,#NULL_SEG

CPB RL6,KST.A_SEG_NO(R13)

IF EQ THEN !MENTOR SEG NOT KNOWN!

LD RO,#MENTOR_SEG_NOT_KNOW

ELSE

PUSH @R15,R13

CALL PROCESS_CLASS IRR2:PROC_CLASS!

IRR4: MENTOR SEG CLASS!

IR1: (RET) CONDITION_CODE!

IR1: (RET) CONDITION_CODE!

POP R13,@R15

LD R6,R1

LDM R1,@R15,#5 IRESTORE INPUT REGS!

CP R6,#FALSE

IF EQ THEN

LDM R1,R15,#5 IRESTORE INPUT REGS!

PUSH @R15,R13 ISAVE ~KST!

LDL RR2,RR4 ICLASS!

LDL RR4,KST.CLASS(R13)

CALL CLASS_GE IRR2:CLASS!

IRR4:M ENTOR CLASS!

IRET:R1:COND_CODE!

IRR4:M ENTOR CLASS!

I RET:R1:COND_CODE!

POP R13,@R15 IRESTORE PTR1

CP R1,#FALSE

LDM R1,@R15,#5

IF EQ THEN

LDM R1,R15,#5

IF EQ THEN

PUSH @R15,R13 ISAVE ~KST!

LOCAL REGS!

- 367 -
ELSE
    LDA R1, KST_MH_HANDLE(R13)
    CALL MM_CREATE_ENTRY
    IR1: PTR TO MM_HANDLE!
    IR2: ENTRY_NO!
    IR3: SIZE!
    IR4: CLASS!
    IR0: (RETURNED) SUCCESS_CODE!
    CALL CONFINEMENT_CHECK
    (RO: SUCCESS_CODE)!
    FI
    FI
ADD R15, #10
    FI
    FI
END CREATE_SEG

- 368 -
**DELETE_SEG**

PROCEDURE

*****************
CHEKS VALIDITY OF DELETE REQUEST AND CALLS MM_DELETE IF VALID.

*****************
REGISTER USE:
PARAMETERS

R1: MENTOR_SEG_NO (INPUT)
R2: ENTRY_NO (INPUT)
R0: SUCCESS_CODE (RET)
LOCAL USE:
R6: VARIOUS LOCAL USES

ENTRY

00A4 93F1 PUSH @R15, R1 !SAVE NEEDED REGS!
00A6 93F2 PUSH @R15, R2
00A8 2101 LD R1, #KST_SEG_NO
00AA 0002
00AC 5F00 CALL ITC_GET_SEG_PTR !R1: KST_SEG_NO!
00AE 0000*
00B0 A10D LD R13, R0 !-KST!
00B2 97F2 POP R2, @R15 !RESTORE INPUT REGS!
00B4 97F1 POP R1, @R15
00B6 0301 SUB R1, #R2:PROCCLASS !CONVERT
MENTOR_SEG_NO TO
00B8 000A

KST REC INDEX!
00BA 1900 MULT RR0, #SIZEOF KST_REC !OFFSET TO DESIRED REC!
00BB 0010
00BE 811D ADD R13, R1 !ADD OFFSET TO KST BASE ADDRESS!
00C0 2106 LD R6, #NULL_SEG
00C2 FFFF
00C4 4ADE CPB RL6, KST_M_SEG_NO(R13)
00C6 000E
00C8 5E0E IF EQ THEN !MENTOR SEGMENT NOT KNOWN!
00CA 00D4
00CC 2100 LD R0, #MENTOR_SEG_NOT_KNOWN
00CE 0016
00D0 5E08 ELSE
00D2 010E*
00D4 93F1 PUSH @R15, R1 !SAVE NEEDED REGS!
00D6 93F2 PUSH @R15, R2
00D8 93FD PUSH @R15, R13
00DA 5F00 CALL PROCESS_CLASS
00DC 0000*

! (RETURNS RR2: PROC_CLASS)!
00DE 97FD POP R13, @R15
00E0 54D4 LDL RR4, KST_CLASS(R13) !MENTOR

- 369 -
SEG CLASS!

00B3 000A  PUSH  @R15,R13
00B6 5F00  CALL  CLASS_EQ  IRR2:PROCESS CLASS!

IRR4:MENTOR SEG CLASS!
IRR1: (RET) CONDITION_CODE!

00C2 A116  LD  R6,R1
00C5 97FD  POP  R13,@R15
00C8 97F2  POP  R2,@R15  IRESTORE NEEDED REGS!
00CE 97F1  POP  R1,@R15
00D1 0B06  CP  R6,#FALSE
00D4 0000
00D5 520E  IF  EQ  THEN
00D8 0102  LD  R0,#ACCESS_CLASS_NOT_EQ
00DC 0021
00DE 5208  ELSE
00E0 0102  LDA  R1,KST.MM_HANDLE(R13)
00E4 0000
00E6 5F00  CALL  MM_DELETE_ENTRY
00E9 0000*  IR1:MM_HANDLE!
00EA 97F2  IR2:ENTRY_NO!
00EF 0000*  IR0:(RET) SUCCESS_CODE!
0100 5F00  CALL  CONFINEMENT_CHECK
0103 0428*  (RO:SUCCESS_CODE) I
0108 9208  FI
010C 0428*  FI
0110 9B08  END DELETE_SEG
0110  MAKE_KNOWN
PROCEDURE

*****************************************************************************
| CHECKS VALIDITY OF MAKE KNOWN |
| REQUEST AND CALLS MM_ACTIVATE |
| IF VALID. ASSIGN SEG |
| NUMBER AND UPDATES KST. |
*****************************************************************************

| REGISTER USE: |
| PARAMETERS: |
| R1:MENTOR_SEG_NO (INPUT) |
| R2:ENTRY_NO (INPUT) |
| R3:ACCESS_DESIRE (INPUT) |
| R0:SUCCESS_CODE (RET) |
| R1:SEGMENT_NO (RET) |
| R2:ACCESS_ALLOWED (RET) |
| LOCAL USE |
| IDENTIFIED AT POINT OF USAGE |
*****************************************************************************

ENTRY

0110 93F1  PUSH @R15, R1  ISAVE INPUT REGS!
0112 91F2  PUSHL @R15, RR2
0114 2101  LD  R1, #KST_SEG_NO
0116 0002
0118 5F00  CALL ITC_GET_SEG_PTR 1 (R1:KST_SEG_NO,  
           RET: R0: -KST) 1
0119 0000*  
011C A1D  LD  R13, R0  !-KST!
011E 95F2  POPL RR2, @R15
0120 97F1  POPO R1, @R15
0122 A115  LD  R5, R1  !COPY OF MENTOR_SEG_NO!
0124 0305  SUB  R5, #R_OF_KSEGS  !CONVERT TO INDEX!
0126 000A  MULT  RR4, #SIZEOF KST_REC !KST OFFSET TO SEG REC!
0128 1904  ADD  R13, R5  ADD OFFSET TO -KST!
012A 0010  
012C 815D  ADD  R13, R5  !ADD OFFSET TO -KST!
012E 2104  LD  R4, #NULL_SEG
0130 FFFF
0132 4ADC  CPB  RR4, KST_M_SEG_NO(R13)
0134 000E  
0136 5E0E  IF EQ THEN
0138 014A*  
013A 2100  LD  R0, #MENTOR_SEG_NOT_KNOWN
013C 0016  
013E 2101  LD  R1, #NULL_SEG
0140 FFFF
0142 2102  LD  R2, #NULL_ACCESS
0144 0004  
0146 5E0E  ELSE
0148 02C8*  
014A 2107  LD  R7, #0 !KST INDEX!
014C 0000  

- 371 -
014E 2108  LD  R8,#NULL_SEG #AVAIL SEG INDEX!
0150  FPPP
0152  A109  LD  R9,R0  #KST!
0154  210A  LD  R10,#NULL_SEG #SEG KNOWN INDICATOR!
0156  FPPP

SEE_IF_KNOWN:

DO
0158  4A99  CPB  RL1,KST.M_SEG_NO(R9)
015A  000E  IF EQ THEN
015C  5E0E  017C'
0160  4A9A  CPB  RL2,KST.ENTRY_NUMBER(R9)
0162  000F  IF EQ THEN  #CASE: SEG KNOWN!
0164  5E0E  017C'
0166  017C'
0168  2100  LD  R0,#SUCCEEDED
016A  0002  ADD  R7,#NR_OF_KSEGS
016C  000A  0107
0170  A171  LD  R1,R7  #SEG#!
0172  609A  LDB  RL2,KST.ACCESS_MODE(R9)
0174  0008  0115
0176  A11A  LD  R10,R1  #SET SEG KNOWN
0178  5E08  EXIT FROM SEE_IF_KNOWN
017A  01A6'  FI

FI
017C  4A9C  CPB  RL4,KST.M_SEG_NO(R9)
017E  000E  #SEE IF SEG # AVAIL!
0180  5E0E  0192'
0182  0192'
0184  0B08  CPB  R8,#NULL_SEG
0186  FPPP
0188  5E0E  IF EQ THEN
018A  0192'
018C  A178  LD  R0,R7  #SAVE FIRST
018E  0108  ADD  R8,#NR_OF_KSEGS
0190  000A  #CONVERT TO SEG #!

FI
0192  A970  INC  R7
0194  0109  ADD  R9,#SIZEOF_KST_REC
0196  0010  #INCREMENT ONE REC!
0198  0807  CP  R7,#MAX_NO_KST_ENTRIES
019A  0036  0109
019C  5E02  IF GT THEN

- 372 -
EXIT FROM SEE_IF_KNOW

PI

OD

!SEE_IF_KNOW!

CP R10,#NULL_SEG

IF EQ THEN ISEG KNOWN

INDICATOR NOT SET!

CP R8,#NULL_SEG

IF ME THEN ICASE SEG UNKNOWN

AND SEG # AVAILABLE!

PUSHL @R15,R20 1-KST AND

MENTOR_SEG_NO1

PUSHL @R15,R2 ENTRY №

ACCESS_DENIED!

PUSH @R15,R8 AVAILABLE SEG

INDEX IN KST!

PUSH @R15,R13 MENTOR SEG REC PTR!

CALL GET_DBG_NUMBER

RET:R11:DBB_NO!

LD R10,R1 IBBW_NO!

LD R13,@R15

POP @R15, R15

POP @R2, @R15

POLL @R2, @R15

POLL @R0, @R15

MUST REARRANGE REGS FOR PASSING AND

RETURN CONSISTENCY OF LOCATION!

LD R5,R3 ACCESS_DENIED!

LD R3,R2 ENTRY №1

LDA R2,KST_MM_HANDLE(R13) IHPTR!

0000

LD R6,R1 MENTOR_SEG_NO1!

LD R1,R8 ISEGMENT № (SAVE)!

LD R4,R8 ISEGMENT №

(PASSING ARG)!

LD R9,R0 1-KST!

SUB R15,#20

SUB R15,#20

LD @R15,R1,#10 I SAVE REGS 1-10!

LD R1,R10 IBBW NO PASSED

IN R1!

LD R11, R8

SUB R11, #NR_OF_KSEGS

MULT RR10,#SIZEOF KST_REC

- 373 -
01F0 A1BC  LD  R12, R11
01F2 819C  ADD  R12, R9
01F4 5F00  CALL  MH_ACTIVATE
01F6 0000*  (R1:DBR_NO, R2:HPTR, R3:ENTRY_NO,
             R4:SEGMENT_NO, R12:RET_HPTR)!
             (RET:R0:SUCCESS_CODE, R2:CLASS,
             R4:SIZE)!
01F8 5F00  CALL  CONFIMENTEN_Check
             (R0:SUCCESS_CODE)!
01FA 0428'  LDL  RR10,RR2  ICLASS1
01FE 942A  LD  R12,R4  ISIZE1
0200 1CF1  LDM  R1,AR15,#9  IRESTORE_REGS 1-9!
0202 0108
0204 A187  LD  R7,R8  ISEG #1
0206 0307  SUB  R7,#R_OF_KSEGS
0208 000A
020A 1906  MULT  RR6,#SIZEOF_KST_REC
             TOffset TO REC1
020C 0010
020E A17D  LD  R13,R7
0210 819D  ADD  R13,R9  IADD -KST TO OFFSET!
0212 50DA  LDL  KST_CLASS(R13),RR10  ICLASS1!
0214 000A
0216 6FD8  LD  KST_SIZE(R13),R12  ISIZE1
0218 0006
021A 0A08  CPB  RLO,#SUCCEEDED
021C 0202
021E 520E  IF  EQ  THEN
0220 02AC*  PUSH  AR15,R13
0222 93FD  CALL  PROCESS_CLASS
0224 5F00  0226 0000*  (RET:RR2:PROC_CLASS)!
0228 97FD  POP  R13,AR15
022A 54D4  LDL  RR4,KST_CLASS(R13)
022C 000A
022E 93FD  PUSH  AR15,R13
0230 91F2  PUSHL  AR15,RR2
0232 19F4  PUSHL  AR15,RR4
0234 5F00  CALL  CLASS_GE
0236 0000*  (R2:PROC_CLASS, R4:SEG_CLASS, RET:
             R1:CONDITION_CODE)!
0238 95F4  POPL  RR4,AR15
023A 95F2  POPL  RR2,AR15
023C 97FD  POP  R13,AR15
023E 0B01  CP  R1,#FALSE
0240 0000
0242 520E  IF  EQ  THEN  NO_ACCESS
             POSSIBLE--DEACT.!
0246 1CF1 LDM R1, R15, #10
0248 0109 LD R1, R10, IDB_NOI
024A A1A1 LDA R2, KST.MM_HANDLE(R13) HPRTR
024C 76D2 CALL MM_DEACTIVATE
0250 5F00 !RET: R0: S_CODE!
0252 0000 CALL CONFINEMENT_CHECK
0254 5F00 !RET: R0: S_CODE!
0256 0428' LD R1, R15, 1SEG #1
0258 21F1 LD R2, #NULL_ACCESS
025A 2102 LD R0, #PROC_CLASS_NOT_GE SEG_CLASS
025C 0004 ELSE
025E 2100 PUSH @R15, R13
0260 0029 CALL CLASS_EQ !(R2:PROC_CLASS,
0262 5E08 RR4:SEG CLASS,
0264 02A8' RET: R1:CONDITION_CODE)!
0266 93FD POP R13, @R15
0268 5F00 LD R0, R1:CONDITION_CODE!
026A 0000 LDM R1, @R15, #9
026C 97FD CP R0, #TRUE
026E A110 IF EQ THEN
0270 1CF1 CP R5, #WRITE
0272 0108 IF EQ THEN
0274 0800 LDB RL2, #WRITE
0276 0001 ELSE
0278 5E0E LDB RL2, #READ
027A 0290' FI
027C 0B05 ELSE
027E 0000 LDB RL2, #READ
0280 5E0E FI
0282 028A' LDB KST.IN_CORE(R13), #FALSE
0284 CA00 ELSE
0286 5E08 LDB KST.IN_CORE(R13), RL6
0288 028C' ELSE
028A CA01 LDB KST.MAX_SEGMENT_NO(R13), RL3
028C 5E08 FI
028E 0292' LDB KST.MAX_SEGMENT_NO(R13), RL3
0290 CA01 FI
02A0 6EDA      LDB  KST.ACCESS_MODE(R13) , RL2
02A2 0008      LD   R0,#SUCCEEDED
                SUCCESS_CODE!
02A4 2100
02A6 0002      FI
02A8 5E08      ELSE
02AA 02B4'      ED
02AC 2101      LD   R1,#NULL_SEG
02AE FFFF
02B0 2102      LD   R2,#NULL_ACCESS
02B2 0004      FI
02B4 010F      ADD  R15, #20
02B6 0014      ELSE
02B8 5E08      ED
02BA 02C8'      ED
02BC 2100      LD   R0,#NO_SEG_AVAIL
02BE 001B
02C0 2101      LD   R1,#NULL_SEG
02C2 FFFF
02C4 2102      LD   R2,#NULL_ACCESS
02C6 0004      FI
              FI
02C8 9B08      RET
02CA          END_MAKE_KNOW
**TERMINATE**  **PROCEDURE**

<table>
<thead>
<tr>
<th>02CA</th>
<th>A113</th>
<th>LD</th>
<th>R3, R1</th>
<th>1COPY OF SEG #1</th>
<th>02CC</th>
<th>0303</th>
<th>SUB</th>
<th>R3, #WB_OF_KSEGS</th>
<th>1CONVERT SEG# TO KST INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>02CE</td>
<td>000A</td>
<td>MULT</td>
<td>R2, #SIZEOF KST_REC</td>
<td>02D0</td>
<td>1902</td>
<td>SUB</td>
<td>R2, #KST_SEG_NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02D2</td>
<td>0010</td>
<td>ADD</td>
<td>R13, R3</td>
<td>02D4</td>
<td>97F1</td>
<td>POP</td>
<td>R15, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02D6</td>
<td>97F3</td>
<td>POP</td>
<td>R15, R3</td>
<td>02D8</td>
<td>2101</td>
<td>LD</td>
<td>R1, #KST_REG_NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02DA</td>
<td>0002</td>
<td>LD</td>
<td>R3, R1</td>
<td>02DC</td>
<td>5F00</td>
<td>CALL</td>
<td>ITC_GET_SEG_PTR</td>
<td>(R7: KST_SEG_NO)</td>
<td></td>
</tr>
<tr>
<td>02DE</td>
<td>0000*</td>
<td>!(RETURNS: R0: KST_SEG_PTR) !</td>
<td>02E0</td>
<td>A10D.</td>
<td>LD</td>
<td>R13, R0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02E2</td>
<td>97F3</td>
<td>POP</td>
<td>R3, R15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02E4</td>
<td>97F1</td>
<td>POP</td>
<td>R1, R15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02E6</td>
<td>813D</td>
<td>ADD</td>
<td>R13, R3</td>
<td>02E8</td>
<td>2106</td>
<td>LD</td>
<td>R6, #NULL_SEG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02FA</td>
<td>FFFF</td>
<td>CPB</td>
<td>RL6, KST_SEG_NO(R13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02BE</td>
<td>000E</td>
<td>IF</td>
<td>EQ</td>
<td>THEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02F2</td>
<td>02FC*</td>
<td>LD</td>
<td>R0, #SEGMENT_NOT_KNOWN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02F4</td>
<td>2100</td>
<td>ELSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02FA</td>
<td>0346*</td>
<td>LD</td>
<td>R6, #FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02FC</td>
<td>2106</td>
<td>IF</td>
<td>EQ</td>
<td>THEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02FE</td>
<td>0001</td>
<td>CPB</td>
<td>RL6, KST_IN_CORE(R13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0300</td>
<td>4ADE</td>
<td>0302</td>
<td>0009</td>
<td>0304</td>
<td>5E08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0306</td>
<td>0310*</td>
<td>IF</td>
<td>EQ</td>
<td>THEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0308</td>
<td>2100</td>
<td>LD</td>
<td>R0, #SEGMENT_IN_CORE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 377 -
ELSE

CP R1,#NR_OP_KSEGS

IF LT THEN

LD R0,#KERNEL_SEGMENT

ELSE

PUSH @R15,R13

CALL GET_DBR_NUMBER

! (RETURNS: BL1:DBR_NO)!

POP R13, @R15

LDA R2, KST.MM_HANDLE (R13)

PUSH @R15, R13

CALL MM_DEACTIVATE ! (R1:DBR_NO)!

! (R2:MM_HANDLE)!

! (RET:R0:SUCCESS_CODE)!

CALL CONFINEMENT_CHECK

! (R0:SUCCESS_CODE)!

POP R13, @R15

CPB RL0,#SUCCEEDED

IF EQ THEN !UPDATE KST

LDB KST.M_SEG_NO(R13)

ENDIF

ENDIF

ENDIF

RET

END TERMINATE
PROCEDURE

I CHECKS VALIDITY OF SNAP IN REQUEST AND CALLS IMSWP_IN IF VALID

I REGISTER USE
I PARAMETERS
I R!:SEGNENT_NO (INPUT)
I RO:SUCCESSCODE (RET)
I LOCAL USE
I R7:KST_INDEX
I R3:ACCESS MODE
I R6:CONSTANT STORAGE

ENTRY
B7:R1 COPY OF SEG #1
SUB R7, #OPESG
I R1:KST_INDEX
I R13:ENTRY

MULT R6,#NULLSEG
LD R6,1FFFFH

IF #KSTSEGNO(R13) THEN
ADD R13,R7
ADD OFFSET TO KST_BASE_ADDR

CPB R6,KST.INCORE(R13)
IF EQ THEN
LD R1,#SEGNENT_NOTKNOWN
ADD R7,R13

ELSE
LD R1,#SUCCESS
ADD R7,R13

END
0388 0002   ELSE
038A 5E08   CALL GET_DBR_NUMBER IR1: (RET) DBR_NO!
038C 0388'  PUSH @R15, R13 ISAVE KST REC ADDR!
038E 93FD   POP R13, @R15
0390 5F00   LDA R2, KST.MH_HANDLE(R13)
0392 0000*  LDB RL3, KST.ACCESS_MODE(R13)
0394 97FD   PUSH @R15, R13 ISAVE SEG KST REC ADDR!
0396 76D2   CALL MM_SWAP_IN IR1: DBR_NO !
0398 0000*   !R2:=MM_HANDLE!
039A 60DB   !R3:=ACCESS_MODE!
039C 0008   !R0: (RET) SUCCESS_CODE!
039E 93FD   CALL CONFINEMENT_CHECK ! (R0: SUCCESS_CODE) !
03A0 5F00   POP R13, @R15
03A2 0000*   CPB RL0, #SUCCEEDED
03A4 5E0E   IF EQ THEN
03A6 0428'   LDB KST.IN_CORE(R13), #TRUE
03A8 97FD   IP
03AA 0A08   FI
03AC 0202   FI
03AE 5E08   FI
03A0 0388'   RET
03BA END MM_SWAP_IN
SM_SWAP_OUT PROCEDURE

!******************************************************************************!
! CHECKS VALIDITY OF SWAP OUT REQUEST AND CALLS IF VALID
!******************************************************************************!
!
ENTRY

03BA A117    LD   R7,R1   _COPY OF SEG #1
03BC 0307    SUB   R7,@NR_OF_KSEGS
                _CONVERT SEG# TO KST INDEX!
03BE 000A    MULT  RR6,SIZEOF_KST_REC
                _OFFSET TO KST_REC!
03C2 0010    PUSH @R15,R1  _SAVE SEGMENT#!
03C4 93F1    PUSH @R15,R7
03C6 93F7    LD   R1,KST_SEG_NO
03CA 0002    CALL ITC_GET_SEG_PTR  IR1:KST_SEG_NO!
03CE 0000*   CALL ITC_GET_SEG_PTR  IR1:KST_SEG_NO!
03D0 A10D    LD   R13,R0  _KST!
03D2 97F7    POP  R7,@R15
03D4 97F1    POP  R1,AR15  _RETRIEVE SEGMENT#!
03D6 817D    ADD  R13,R7  _ADD OFFSET TO KST BASE ADDR!
03D8 2106    LD   R6,NULL_SEG
03DA 0000    CPB  RL6,KST.M_SEG_NO(R13)
03DE 000E    IF EQ THEN
03E0 520E    IF EQ THEN
03E2 03EC*   IF EQ THEN
03E4 2100    LD   R0,SEGMENT_NOT_KNOWN
03E6 001C    ELSE
03EB 0014    ELSE
03EC 0426*   ELSE
03EE 2106    LD   R6,FALSE
03EF 0000    CPB  RL6,KST.IN.Core(R13)
03F2 0009    IF EQ THEN
03F4 520E    IF EQ THEN
03F6 0000*   IF EQ THEN
03F8 2100    LD   R0,SUCCEEDED
03FA 0002 ELSE
03FC 5E08
03FE 0426' PUSH @R15,R13 ISAVE KST REC ADDR1
0400 93FD CALL GET_DB_NUMBER 1R1:(RET)DBR_NO1
0404 0000* POP R13,@R15
0406 97FD LDA R13,KST.MM_HANDLE(R13)
0408 76D2 CALL MRSUAP-OUT 1R1:DBR_NO1
040A 0000* POP R13,@R15
040C 93FD CALL MM_SWAP_OUT 1R1:DBR_NO1
0410 0000* !R2:MM_HANDLE!
0412 5E0E CALL CONFINEMENT_CHECK
0414 0428'quire (RET)SUCCESS_CODE!
0416 97FD !R0:(RET)SUCCESS_CODE!
0418 0A08 CPB RL0,SUCCEEDED
041A 0202 IF EQ THEN
041C 5E0E LDB KST.I4_Core(R13),#FALSE
0420 4CD5 PI
0422 0009 PI
0424 0000 PI
0426 9E08 RET
0428 END SN_SWAP_OUT

- 382 -
CONFINEMENT_CHECK PROCEDURE

SERVICE ROUTINE TO VERIFY
CONFINEMENT IS NOT VIOLATED
WHEN MGR SUCCESS_CODE IS
RETURNED TO SUPERVISOR.

REGISTER USE:
PARAMETERS
RO:SUCCESS_CODE

ENTRY

IF RO
CASE #LEAF_SEG_EXISTS THEN
CALL MONITOR

CASE #NO_LEAF_EXISTS THEN
CALL MONITOR

CASE #ALIAS_DOES_NOT_EXIST THEN
CALL MONITOR

CASE #NO_CHILD_TO_DELETE THEN
CALL MONITOR

CASE #G_AST_FULL THEN
CALL MONITOR

- 383 -
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

END CONFINEMENT_CHECK

END SEG_MGR
Appendix I

NON-DISCRETIONARY SECURITY LISTINGS

28000ASM 2.02
LOC OBJ CODE STMT SOURCE STATEMENT

$LISTON $TTY
NDS MODULE

CONSTANT
 TRUE :=1
 FALSE :=0

INTERNAL
$SECTION ACC_CLASS_DCL !NOTE: IS AN OVERLAY, IE NO ALLOCATION
OF MEMORY!

$ABS 0

0000 ACCESS_CLASS RECOED [LEVEL INTEGER
CAT INTEGER]

GLOBAL
$SECTION NDS_PROC

0000 CLASS_EQ PROCEDURE
[*******************]
! PASSED PARAMETERS
! RR2 = CLASS1
! RR4 = CLASS2
! RETURNED
! R1 = CONDITION_CODE
[*******************]

ENTRY
0000 9042 CPL RR2,RR4
0002 5E0E IF EQ THEN
0004 000E'
0006 2101 LD R1,#TRUE
0008 0001
000A 5E08 ELSE
000C 0012'
000E 2101 LD R1,#FALSE
0010 0000 FI
0012 9E08 RET
0014 END CLASS_EQ

- 385 -
CLASS_GE

PROCEDURE

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

PASSED PARAMETERS

R2 = CLASS1

R4 = CLASS2

RETURNED PARAMETER

R1 = CONDITION CODE

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

ENTRY

0014 91F2  PUSHL @R15,RR2  IPUSH CLASS1 ON STACK REFER BY ADDR

0016 A1FD  LD  R13,R15  I CLASS1 ADDR 1

0018 91F4  PUSHL @R15,RR4

001A A1FE  LD  R14,R15  I CLASS2 ADDR 1

001C 31E7  LD  R7,R14 (#ACCESS_CLASS.CAT)

I CAT2 IN R7

001E 0002  OR  R7,ACCESS_CLASS.CAT(R13)

I CAT1 OR CAT2, R7

0020 45D7  CP  R7,ACCESS_CLASS.CAT(R13)

I CAT1=(CAT1 OR CAT2)?

0022 0002  IF EQ THEN

0024 4BD7  CP  R7,ACCESS_CLASS.CAT(R13)

I CAT1=(CAT1 OR CAT2)?

0026 0002  ELSE

0028 5E0E  IF EQ THEN

002A 0048  ELSE

002C 61D6  LD  R6,ACCESS_CLASS.LEVEL(R13)

I LEVEL1

002E 0000  END IF

I COMPARE LEVEL1 WITH LEVEL2

0030 4B86  CP  R6,ACCESS_CLASS.LEVEL(R14)

I LEVEL2

0032 0000  IF GE THEN I LEVEL1 GE LEVEL2

0034 5E01  LD  R1,#TRUE

0036 0040  ELSE

0038 2101  LD  R1,#FALSE

003A 0001  END IF

003C 5E08  ELSE

0040 2101  LD  R1,#FALSE

0042 0000  FI

0044 5E08  ELSE

0046 004C  END IF

0048 2101  LD  R1,#FALSE

004A 0000  PI

004C 95F4  POPL RR4,@R15

004E 95F2  POPL RR2,@R15 I RESTORE STACK1

0050 9E08  RET

0052  END CLASS_GE

END NDS

- 386 -
Appendix J

MEMORY MANAGER LISTINGS

Z8000ASM 2.02

LOC OBJ CODE STMT SOURCE STATEMENT

$LISTON sty
MN_PROCESS MODULE

! VERS. 1.9 !

CONSTANT
RETURN_TO_MONITOR := %A902 !BUG REENTRY!

COUNT := 10
TIME := 500

NR_OF_HOSTS := 2
G_AST_LIMIT := 10
G_AST_PULL := 12
FREE_ENTRY := %E2E2E2E2
FALSE := %C2C2C2C2
SPACE := %20
DASH := %2D

IO_MGR := %60
FILE_MGR := %40
MEM_MGR := %00
FM_ENTRY := %4A00
IO_ENTRY := %4E00
CREATE_ENTRY_CODE := 50
INVALID_MGR_CODE := 60
DELETE_ENTRY_CODE := 51
ACTIVATE_SEG_CODE := 52
DEACTIVATE_SEG_CODE := 53
SWAP_IN_SEG_CODE := 54
SWAP_OUT_SEG_CODE := 55
SUCCEEDED := 2
STK_SIZE := 1
TOP_SECRET := 4
SECRET := 3
CONFIDENTIAL := 2
UNCLASS := 1
EMPTY := 0
CRYPTO := 1

- 387 -
NATO := 2
NUCLEAR := 4

TYPE
ADDRESS WORD
H_ARRAY ARRAY[3 WORD]

G_AST_REC RECORD
[ UNIQUE_ID LONG
GLOBAL_ADDR ADDRESS
P_LASTE_NO WORD
FLAG_BITS WORD
G_AST_PAR WORD
NO_ACT_IN_MEM WORD
NO_ACT_DEP BYTE
SIZE1 BYTE
PG_TBL_LOC ADDRESS
ALIAS_TBL_LOC ADDRESS
SEQUENCER LONG
EVENT1 LONG
EVENT2 LONG
]

EXTERNAL
SIGNAL PROCEDURE
WAIT PROCEDURE
TC_INIT PROCEDURE
GET_CPU_NO PROCEDURE
CREATE_PROCESS PROCEDURE
SNDCHR PROCEDURE
SNDMSG PROCEDURE
SNDCLP PROCEDURE

G_AST_LOCK WORD
G_AST ARRAY[G_AST_LIMIT G_AST_REC]

GLOBAL
$SECTION MM_DATA
MM_ENTRY LABEL

- 388 -
INTERNAL

! ** * * * MESSAGES * * * * !
0000 08 28  IO  ARRAY [* BYTE] := '%08(FOR IO)'
0002 46 4F
0004 52 20
0006 49 4F
0008 29
0009 08 28  FM  ARRAY [* BYTE] := '%08(FOR FM)'
000B 46 4F
000D 52 20
000F 46 4D
0011 29
0012 12 4B  MM_MSG_1
  ARRAY [* BYTE] := '%12KERNEL = SIGNALLER'
0014 45 52
0016 4E 45
0018 4C 20
001A 3D 20
001C 53 49
001E 47 48
0020 45 4D
0022 4C 45
0024 52
0025 10 4D  CREATE_MSG
  ARRAY [* BYTE] := '%10MM: CREATE_ENTRY'
0027 4D 3A
0029 20 43
002B 52 45
002D 41 54
002F 45 5F
0031 45 4E
0033 54 52
0035 59
0036 10 4D  DELETE_MSG
  ARRAY [* BYTE] := '%10MM: DELETE_ENTRY'
0038 4D 3A
003A 20 44
003C 45 4C
003E 45 54
0040 45 5F
0042 45 4E
0044 54 52
0046 59

- 389 -
ACTIVATE_MSG
ARRAY [* BYTE] := 'OCMM: ACTIVATE'

DEACTIVATE_MSG
ARRAY [* BYTE] := 'OCMM: DEACTIVATE'

SWAP_IN_MSG
ARRAY [* BYTE] := 'OCMM: SWAP_IN'

SWAP_OUT_MSG
ARRAY [* BYTE] := 'OCMM: SWAP_OUT'

ERROR_MSG
ARRAY [* BYTE] := 'OCINVALID CODE'

RET_VALUES
ARRAY [* BYTE] := [2, 0, 0, 0, 0, 16, 0, 17, 0, 3, 0, 1, 0, 48, 0, 0]

MM_MSG_ARRAY ARRAY [ 8 WORD ]
SENDER WORD

- 390 -

0047 OC 4D 0049 4D 3A 004B 20 41 004D 43 54 004E 49 56 0050 41 54 0051 41 54 0052 41 54 0053 41 54 0054 02 4D 0056 4D 3A 0058 20 44 005A 45 41 005C 43 54 005E 49 3A 0060 41 54 0062 45 41 0063 0B 4D 0065 4D 3A 0067 20 44 0069 57 41 006B 50 5F 006D 49 4E 006F 0C 4D 0071 4D 3A 0073 20 53 0075 57 41 0077 50 5F 0079 4F 55 007B 54 007C 0C 49 007E 4E 56 0080 41 4C 0082 49 44 0084 20 43 0086 4F 44 0088 45 0089 02 00 009B 00 00 009D 00 10 009F 00 11 0091 00 03 0093 00 01 0095 00 30 0097 00 00 099A MM_MSG_ARRAY ARRAY [ 8 WORD ] 00AA SENDER WORD
$ABS 0
NO MEMORY ALLOCATED; USED FOR PARAMETER TEMPLATE ONLY!

0000
ACTIVATE_ARG RECORD
[ CODE WORD
 DBR WORD
 HANDLE H_ARRAY
 ENTRY_NO BYTE
 SEG_NO BYTE
]

$ABS 0
NO MEMORY ALLOCATED; USED FOR PARAMETER TEMPLATE ONLY!

0000
RET_VAL RECORD
[ CODE1 BYTE
 FILLER BYTE
 MM_HANDLE H_ARRAY
 CLASS LONG
 SIZE WORD
 FILLER1 WORD
]

$ABS 0
ARG_LIST RECORD
0000
[ REG ARRAY[13 WORD]
 IC WORD
 CPU_ID WORD
 SAC LONG
 PRI WORD
 USR_STK WORD
 KER_STK WORD
]

- 391 -
$SECTION MN_PROC

0000

PROCEDURE

ENTRY

MN_ENTRY:

I INITIALIZE G_AST I

0000 4D08
CLR G_AST_LOCK
0002 0000*
0004 2102
LD R2, #1
0006 0001
0008 2101
LD R1, #0
000A 0000
000C 1404
LDL RR4, #FREE_ENTRY
000E EE0E
0010 EE0E
DO
0012 5D14
LDL G_AST.UNIQUE_ID(R1), RR4
0014 0000*
0016 A920
INC R2, #1
0018 0B02
CP R2, #G_AST_LIMIT
001A 000A
001C 5E02
IF GT !END OF G_AST! THEN
001E 0024*
0020 5E08
EXIT PI
0022 002A*
0024 0101
ADD R1, #SIZEOF G_AST_REC
0026 0020
0028 E8F4

I RESERVE FIRST ENTRY IN G_AST FOR ROOT I

002A 2101
LD R1, #0
002C 0000
002E 1404
LDL RR4, #-1
0030 FFPP
0032 FFPP
0034 5D14
LDL G_AST.UNIQUE_ID(R1), RR4
0036 0000*
0038 5F00
CALL GET_CPU_NO !RETURNS:
R1: CPU #
R2: # VP'S!
003A 0000*
003C 93F1
PUSH bR15, R1 !SAVE CPU #1
003E 5F00
CALL TC_INIT
0040 0000*

I USER/HOST # I

0042 210D
LD R13, #0
0044 0000
I INITIALIZE USERS I

DO
0046 A9D0
INC R13, #1

- 392 -
0048 0B0D CP R13, #NR_OF_HOSTS
004A 0002 IF GT !ALL HOSTS INITIALIZED!
004C 5E02 THEN EXIT
004E 0054 FI
0050 5E08 ! CREATE FM PROCESS !
0052 00B8 0054 21F0 LD R0, @R15 I RESTORE CPU #1
0056 030F SUB R15, #SIZEOF ARG_LIST
0058 0028 ! SETS ARGUMENT LIST IN STACK! 005A A1F1 LD R1, R15
005C 6F10 LD ARG_LIST.CPU_ID(R1), R0
005E 001C ! LOAD INITIAL REGISTER PARAMETERS FOR FM PROCESS (SIMULATED) R13 DENOTES USER # 1
0060 5C19 LDM ARG_LIST.REG(R1), R R2, #13
0062 020C 0064 0000 0066 2102 LD R2, #FM_ENTRY
0068 4A00 006A 6F12 LD ARG_LIST.IC(R1), R2
006C 001A 006E 2102 LD R2, #SECRET
0070 0003 0072 8D38 CLR R3
0074 0503 OR R3, #CRYPTO
0076 0001 0078 5D12 LDL ARG_LIST.SAC(R1), RR2
007A 001E 007C 4D15 LD ARG_LIST.PRI(R1), #2
007E 0022 0080 0002 0082 4D15 LD ARG_LIST.USR_STK(R1), #STK_SIZE
0084 0024 0086 0001 0088 4D15 LD ARG_LIST.KER_STK(R1), #STK_SIZE
008A 0026 008C 0001 008E A11E LDL R14, R1
0090 93FD PUSH @R15, R13
0092 5F00 CALL CREATE_PROCESS !R14: ARG PTR!
0094 0000 POP R13, @R15
0096 97FD ! CREATE IO PROCESS !

- 393 -
LOAD INITIAL REGISTER PARAMETERS
FOR IO PROCESS (SIMULATED)
R13 DENOTES USER # 1

LDM ARG_LIST.REG(R1), R2, #13

LD R2, #IO_ENTRY

LD ARG_LIST.IC(R1), R2

PUSH @R15, R13

CALL CREATE_PROCESS 1R14: ARG PTR

ADD R15, #SIZEOF ARG_LIST

POP R13, @R15

DO !** DO FOREVER **!

LDA R8,MM_MSG_ARRAY 0

CALL WAIT

LD SENDER, R1 !SAVE SIGNALING PROC #!

LD R3,#50

CALL MM_PRINT_BLANKS

LD R2,#MM_MSG_1

CALL SNDMSG

LD R1, SENDER

IF R1

CASE #IO_MGR THEN LD R2,#ID

CALL SNDMSG

CASE #FILE_MGR THEN LD R2,#FM
CALL SNDMSG
CALL MM_DELAY
CALL SNDCLF
LD R3,#50
LD R1,MM_MSG_ARRAY 0
IF  R1 THEN
    CASE #CREATE_ENTRY_CODE THEN
        CALL CREATE_ENTRY
    CASE #DELETE_ENTRY_CODE THEN
        CALL DELETE_ENTRY
    CASE #ACTIVATE_SEG_CODE THEN
        CALL ACTIVATE
    CASE #DEACTIVATE_SEG_CODE THEN
        CALL DEACTIVATE
    CASE #SWAP_IN_SEG_CODE THEN
        CALL SWAP_IN
    CASE #SWAP_OUT_SEG_CODE THEN
        CALL SWAP_OUT
    ELSE
        LD  R2,#ERROR_MSG
        FI
    - 396 -
CALL SNDMSG
CALL MM_DELAY
CALL SNDCLLF
CALL MM_PRINT_LINE
CALL SNDCLLF
CALL SIGNAL
LD R3,#75
LD R1,SENDER
LDA R8,MM_MSG_ARRAY 0
LDA R8.#SUCCEEDED
LD R2,#CREATE_MSG
LD R2,#DELETE_MSG
RET
END MM_MAIN
CREATE ENTRY
ENTRY
LDA R8,MM_MSG_ARRAY 0
LDB @R8,#SUCCEEDED
LD R2,#CREATE_MSG
RET
END CREATE ENTRY
DELETE ENTRY
ENTRY
LDA R8,MM_MSG_ARRAY 0
LDB @R8,#SUCCEEDED
LD R2,#DELETE_MSG
RET
END DELETE_ENTRY
- 397 -
OlBA

ACTIVATE

PROCEDURE

ENTRY

01BA 7608 LDA R8, MM_MSG_ARRAY 0
01BC 009A' LD R2, ACTIVATE_ARG.HANDLE_2 (R8)
01BE 6182 LD R3, ACTIVATE_ARG_ENTRY_NO (R8)
01C0 0008 CLR B3
01C2 8D38 LDB BL3, ACTIVATE_ARG_ENTRY_NO (R8)
01C6 000A SUB R15, #SIZEOF RET_VAL
01CA 0010 CLR B1
01CC A1F8 LD R8, R15
01CE 2100 LD R0, #FALSE
01D0 CCCC
01D2 2101 LD R1, #0 IG_AST_INDEX
01D4 0000
01D6 2104 LD R4, #1 IXR OF ENTRIES SEARCHED
01D8 0001

SEARCH_G_AST:

DO

01DA 5012 CPL RR2, G_AST.UNIQUE_ID (R1)
01DC 0000' IF EQ ISEGMENT IS ACTIVE! THEN
01E0 01EA' LD RO, #TRUE
01E2 2100 IF GT IEND OF G_AST! THEN
01E4 3BBB EXIT FROM SEARCH_G_AST
01E6 5208 ADD R1, #SIZEOF G_AST_REC
01E8 01FE' FI
01EA A940 INC R4, #1
01EC 0B04 CP R4, #G_AST_LIMIT
01EE 000A IF GT IEND OF G_AST! THEN
01F0 5E02 ADD R1, #SIZEOF G_AST_REC
01F2 01F8' FI
01F4 5E08 EXIT FROM SEARCH_G_AST
01F6 01FE' FI
01F8 0101 ADD R1, #SIZEOF G_AST_REC
01FA 0020 CP RO, #FALSE
01FC 88EE OD
01FE 0B00 CP RO, #FALSE
0200 CCCC IF EQ ISEGMENT NOT ACTIVE! THEN
0202 5E0E LD R0, #1
0204 0266' LD R0, #0
0206 2100 LD R1, #0
0208 0001 LD R1, #0

- 398 -
FIND_FREE_ENTRY:
DO
  LDL RR4, #FREE_ENTRY
  CPL RR4, G_AST.UNIQUE_ID(R1)
  IF EQ ENTRY IS AVAILABLE THEN
    EXIT FROM FIND_FREE_ENTRY
  FI
  INC B0, #1
  CP B0, #G_AST_LIMIT
  IF GT END OF G_AST THEN
    EXIT FROM FIND_FREE_ENTRY
  FI
  ADD R1, #SIZEOF_G_AST_REC
  OD
  CP B0, #G_AST_LIMIT
  IF LE FOUND FREE ENTRY THEN
    LDL G_AST.UNIQUE_ID(R1), RR2
    ! ZERO ALL EVENT DATA ENTRIES!
    LDL RR4, #0
    LDL G_AST.SEQUENCES(R1), RR4
    LDL G_AST.EVENT1(R1), RR4
    LDL G_AST.EVENT2(R1), RR4
    LDB RET_VAL.CODE1(R8), #SUCCEEDED
  ELSE
    LDB RET_VAL.CODE1(R8), #G_AST_FULL
  FI
  ELSE SEGMENT ACTIVE!
- 399 -
0264 026C'  LDB  RET_VAL.CODE1 (R8), #SUCCEEDED
0266 4C85  FI
0268 0000  026A 0202
026C 5D82  LDL  RET_VAL.MMHANDLE 0 (R8), R R2
026E 0002  0270 6F81  LD  RET_VAL.MMHANDLE 2 (R8), R1
0272 0006  0274 1404  LDL  R R4, #%30001
0276 0003  0278 0001 027A 5D84  LDL  RET_VAL.CLASS (R8), R R4
027C 0008  027E 4D85  LD  RET_VAL.SIZE (R8), #1
0280 000C  0282 0001 0284 7689  LDA  R9, RET_VAL(R8)
0286 0000  0288 7608  LDA  R8, MM.MSG ARRAY 0
028A 009A' 028C 2102  LDL  R2, #16
028E 0010  0290 BA91  LDS  AR8, AR9, R2
0292 0280  0294 2102  LD  R2, #ACTIVATE_MSG
0296 0047' 0298 010F  ADD  R15, #SIZEOF RET_VAL
029A 0010  029C 9E08  RET 029E  END ACTIVATE
029E  DEACTIVATE  PROCEDURE
ENTRY
029E 7608  LDA  R8,MM_MSG_ARRAY 0
02A0 009A1`
02A2 0C85  LDB  @R8,#SUCCEEDED
02A4 0202
02A6 2102  LD  R2,#DEACTIVATE_MSG
02A8 00541`
02AA 9E08  RET
02AC  END DEACTIVATE

02AC  SWAP_IN  PROCEDURE
ENTRY
02AC 2102  LD  R2,#%FF30
02AE FF30
02B0 3B26  OUT  %FFD2, R2
02B2 FFD2
02B4 7608  LDA  R8, MM_MSG_ARRAY
02B6 009A1`
02B8 5F00  CALL  WAIT !R8:MSG ARRAY!
02BA 00001`
02BC 7608  LDA  R8, MM_MSG_ARRAY 0
02BE 009A1`
02C0 0C85  LDB  @R8,#SUCCEEDED
02C2 0202
02C4 2102  LD  R2,#SWAP_IN_MSG
02C6 00631`
02C8 9E08  RET
02CA  END SWAP_IN
SWAP_OUT

ENTRY

LDA R8, MM_MSG_ARRAY 0
LDB @R8, @SUCCEEDED
LD R2, #SWAP_OUT_MSG
RET

MM_DELAY

PROCEDURE

***** PRODUCES 2 SEC DELAY *****

ENTRY

LD R2, #COUNT
LD R1, #TIME
DO
CP R2, #0
IF EQ THEN EXIT FI
DEC R2
MREQ R1
OD
RET

END MM_DELAY
02F4       MM_PRINT_LINE        PROCEDURE
[********************************************]
| PRINTS LINE LENGTH                      |
| SPEC IN R3.                              |
[********************************************]
ENTRY
02F4 C82D LDB   BLO, #DASH
DO
02F6 0B03   CP   R3, #0
02F8 0000
02FA 5E0E   IF EQ THEN EXIT FI
02FC 0302'
02FE 5E08
0300 030A'
0302 5F00   CALL SNDCHR
0304 0000*
0306 AB30   DEC   R3
0308 E8F6   OD
030A 9E08   RET
030C       END MM_PRINT_LINE

030C       MM_PRINT_BLANKS        PROCEDURE
[********************************************]
| PRINTS NUMBER OF                        |
| BLANKS SPEC IN R3.                      |
[********************************************]
ENTRY
030C C820 LDB   BLO, #SPACE
DO
030E 0B03   CP   R3, #0
0310 0000
0312 5E0E   IF EQ THEN EXIT FI
0314 031A'
0316 5E08
0318 0322'
031A 5F00   CALL SNDCHR
031C 0000*
031E AB30   DEC   R3
0320 E8F6   OD
0322 9E08   RET
0324       END MM_PRINT_BLANKS
            END MM_PROCESS
LIST OF REFERENCES


<table>
<thead>
<tr>
<th>No.</th>
<th>Distribution List</th>
</tr>
</thead>
</table>
| 1.  | Defense Documentation Center  
|     | ATTN: DDC-TC  
|     | Cameron Station  
|     | Alexandria, Virginia  22314 |
| 2.  | Library, Code 0142  
|     | Naval Postgraduate School  
|     | Monterey, California  93940 |
| 3.  | Department Chairman, Code 52  
|     | Department of Computer Science  
|     | Naval Postgraduate School  
|     | Monterey, California  93940 |
| 4.  | National Security Agency  
|     | Attn.: Col. Roger R. Schell  
|     | C1  
|     | Fort George Meade, Maryland  20755 |
| 5.  | Lyle A. Cox, Jr., Code 52C1  
|     | Department of Computer Science  
|     | Naval Postgraduate School  
|     | Monterey, California  93940 |
| 6.  | Joel Trimble, Code 221  
|     | Office of Naval Research  
|     | 800 North Quincy  
|     | Arlington, Virginia  22217 |
| 7.  | John P.L. Woodward  
|     | The MITRE Corporation  
|     | P.O. Box 208  
|     | Bedford, Massachusetts  01730 |
| 8.  | Digital Equipment Corporation  
|     | Attn: Mr. Donald Gaubatz  
|     | 146 Main Street  
|     | ML 3-2/E41  
|     | Maynard, Massachusetts  01754 |

No. Copies
---
2
2
2
5
4
1
1
1
9. Joe Urban
   University of Southwestern Louisiana
   P.O. Box 44330
   Lafayette, Louisiana  70504

10. LCDR Gary Baker, Code 37
    COMMSPATWINGPAC
    Code 32
    NAS Moffett Field, California  94035

11. LCDR John T. Wells
    P.O. Box 366
    Waynesboro, Mississippi  39367

    Box 42
    Fort Washington, Pennsylvania  19034

13. I. Larry Avrunin, Code 18
    DTNSRDC
    Bethesda, Maryland  20084

14. Gerald B. Blanton
    242 San Carlos Way
    Novato, Calif.  94947

15. Intel Corporation
    Attn: Mr. Robert Childs
    Mail Code: SC4-490
    3065 Bowers Avenue
    Santa Clara, California  95051

16. Dr. J. McGraw
    U.C. - L.L.L. (1-794)
    P.O. Box 808
    Livermore, California  94550

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

18. Robert Montee
    Director, Long Range Systems Planning
    Honeywell, MN 12-2276
    Honeywell Plaza
    Minneapolis, Minn.  55408
19. Dr. F. C. Colon Osorio
Research and Development
Digital Equipment Corp.
146 Main Street
Maynard, Mass. 01754

20. David P. Reed
MIT Lab for Computer Science
545 Tech Sq
Cambridge, Mass. 02139

21. Capt. Anthony A. Strickler
HQ Command,
US Army Signal Center & Ft. Gordon (WOU5AA)
Ft. Gordon, Georgia 30905

22. Lt. W. J. Wasson
Naval Electronics Systems Command
Headquarters, PHE 124
Washington D. C. 20360

23. S. H. Wilson
Code 7590
Naval Research Lab
Washington D. C. 20375

24. Maj. Robert Yingling
Box 6227
APO New York, New York 09012

25. LCDR William R. Shockley
Code 52Sp
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

26. LCDR Ronald W. Modes
Code 52Mf
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

27. S. L. Perdue
Code 52
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940