18

HIMICS: A VIRTUAL MEMORY ENVIRONMENT FOR MINI-COMPUTERS AND A DESCRIPTION OF ITS LEVEL 1 PROCESSOR

by

DOUGLAS EUGENE SMITH

B.S., Kansas State University, 1973

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1975

Approved by:

Virgil E. Wallentine

Major Professor

LD
2668
R4
1975
S64
c.2
Document 1

TABLE OF CONTENTS

ILLUSATRATIONS

ILLEGIBLE
DOCUMENT

THE FOLLOWING
DOCUMENT(S) IS OF
POOR LEGIBILITY IN
THE ORIGINAL

THIS IS THE BEST
COPY AVAILABLE

P-474

CHAPTER ONE

1.1 INTRODUCTION

In this paper we propose a design for a hierarchical mini-computer system called HIMICS (Hierarchical multI-tasking MinI-computer Computer System). The system is designed with five major objectives in mind. These objectives are:

(1) To provide virtual memory capability.

(2) To provide support for emulators.

(3) To provide inter-emulator communication.

(4) To provide manageable software.

(5) To provide sufficient instrumentation and monitor capabilities in order to encourage meaningful system evaluations and comparisons.

A general discussion of each of these objectives will be given before we present the actual design of the system.

1.2 TECHNIQUES FOR RECURRENT USE OF MEMORY

There are several techniques that are commonly used in modern day computers to execute programs which have a larger address space than the primary memory available to them. Two of the more commonly used techniques are overlay structures (1,6,12) and virtual memory (2,12).

1.2.1 OVERLAY STRUCTURES

With overlay structures, segments of the program are kept on secondary storage and brought into main memory in an hierarchical sequence as they are needed. Pre-specified segments may be overwritten by incoming segments. This is illustrated in Figure 1-1. Here the user has a 520K program to be run in a 320K address space. Segments A, B and C are first loaded and execution begins. As soon as segment B is no longer needed, segments D and E can be overwritten in B's address space. The same process happens when C and E are no longer needed. They can be overwritten by F.

From this illustration it is apparent that the user must have a knowledge as to what segments are to be overwritten. It is the user's responsibility to issue orders for the overlay to occur. This is the major disadvantage of the overlay technique. The second technique, virtual memory, does not require the user to have this additional knowledge.

1.2.2 VIRTUAL MEMORY

The key to virtual memory relies on the fact that, for an instruction in a program to be executed, only the instruction and the data that it operates on need be in primary memory. From the instruction's point of view the rest of the program may be located on any level of memory. This removes the requirement that the job's entire address space be in physical memory at once. Because this physical restraint is removed, the operating program has the illusion that it has an extremely large memory, thus the term "virtual memory". Since a job's entire address space need not all be in primary memory at once,

Overlay Structure

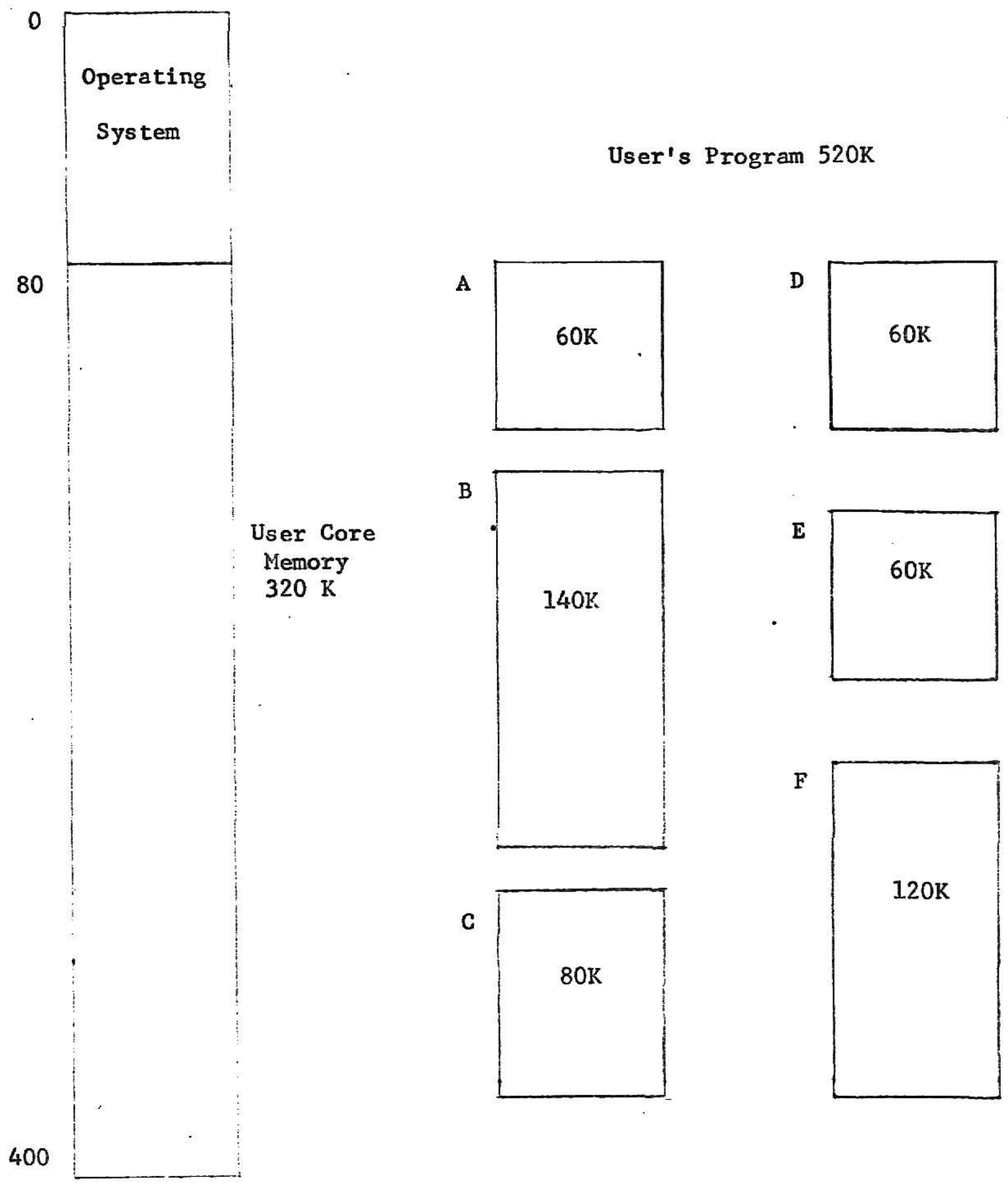


Figure 1-1

the sum of the address spaces of the jobs being multiprogrammed is permitted to exceed the physical size of main memory. This is illustrated in Figure 1-2. All three jobs are being executed in a physical memory space of 280K. The total sum of all three jobs is 460K.

The major constraint as to the size of the virtual space is limited by the hardware configuration. The hardware limits the number of addressable cells. The limiting factor is the number of bits used for an address. For example, if the hardware allows 8 bits for an address, then there are 256 addressable cells. The addresses would range from 0 to 255. This limits the virtual memory to this same size of space. The virtual space is usually considerably larger than the available primary memory of the machine.

1.2.2.1 VIRTUAL MEMORY TECHNIQUES

Two major virtual memory techniques are, demand-paged memory management (1,7) and segmented memory management (1). As was stated previously, virtual memory requires that the instruction and data to be operated on be located in primary memory. If this were to be done one instruction at a time it would be too time consuming. Instead they are retrieved in sections upon demand. If the sections are all equally divided into the same length, they are individually referred to as a page. This is the orgin of the term "demand-paged" memory.

Virtual Memory Structure

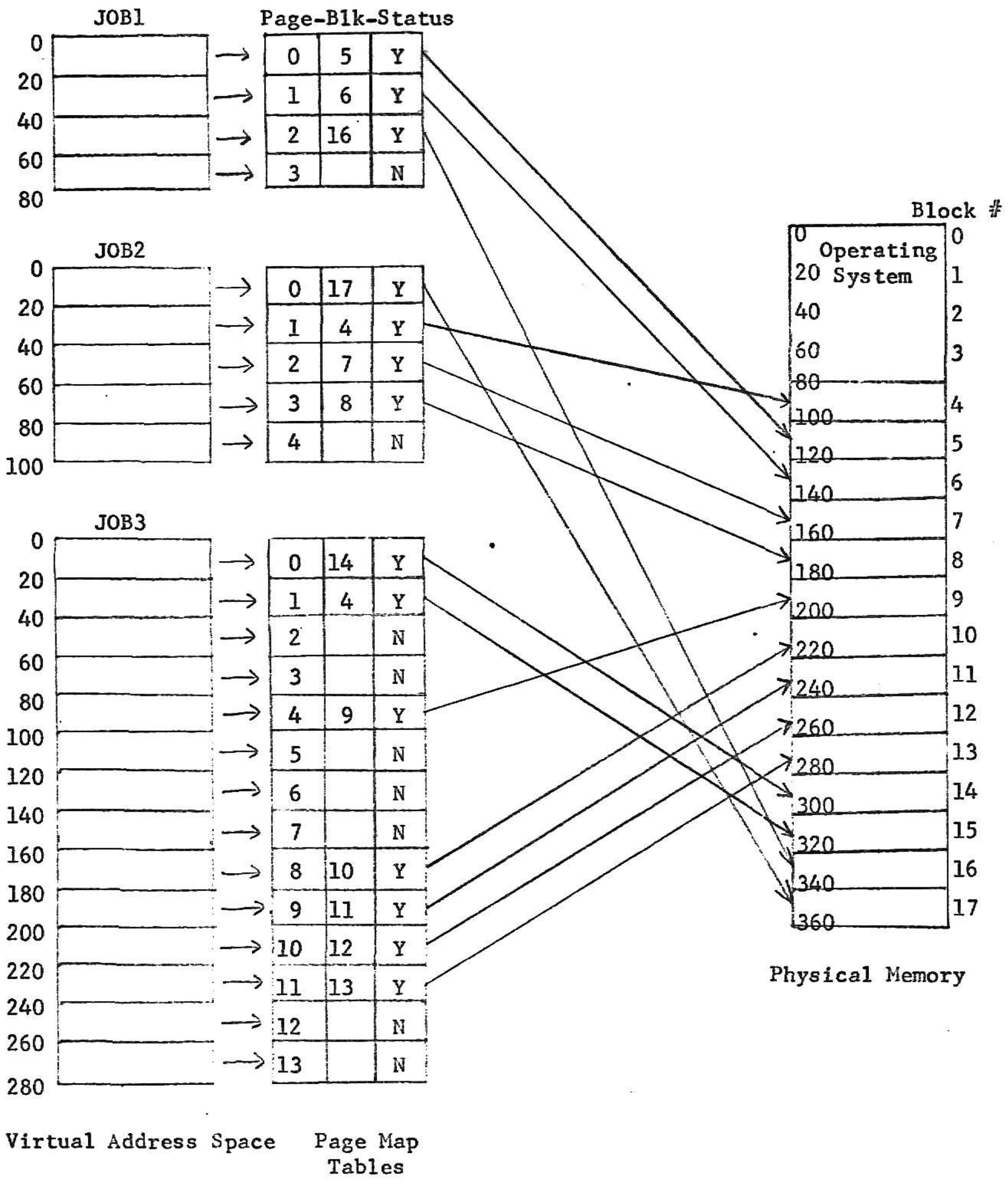


Figure 1-2

If the sections are unequal in length they are called segments, hence "segmented" memory. This paper will only be concerned with the former, demand-paged memory.

At the start of execution of a user's program, the first page is brought into primary memory. This is done by the virtual memory module, which will be explained in sextion 1.7.2. As each instruction is executed, the virtual memory module checks to make sure that all the address space referenced is in primary memory. If the address space is not in primary memory an interrupt, called a page fault (2), is generated. The operating system then processes this interrupt. This is done by loading the required page into primary memory. The process is then restarted from the point of the interrupt. Each additional required page is brought into primary memory upon request.

This can obviously lead to the point where primary memory is full when a new page is being requested by the virtual memory module to be brought into primary memory. To alleviate this situation page replacement (1) is necessary. This consists of removing from primary memory a page that does not have a high probability of being referenced in the near future. This page is then placed in secondary memory while the newly requested page is moved into primary memory.

1.2.2.2 ADVANTAGE OF VIRTUAL MEMORY

Virtual memory is commonly used today by many large computers. There are many advantages to be gained by using virtual memory. Some of these advantages include:

(1) Increase in the number of programs that can be multiprogrammed in a system.

(2) Capability of running a program whose address space exceeds the primary memory space currently available, if less than the maximum addressable memory.

(3) Makes programs more portable from large machines to small machines.

(4) Helps eliminate fragmentation of dynamic storage allocation.

These are especially appealing to mini-computers since mini-computers by nature have a smaller primary memory space.

1.3 SUPPORT FOR EMULATORS

For use in this paper, we will define an emulator to be a firmware interpreter. This interpreter will convert a user program from the original language, instruction by instruction, into the desired computer actions. An emulator written for this system must be aware of the manner in which to access the virtual addressing system of the host processor. An emulator will not create a machine language program. Instead, it will in effect execute a small microprogram for each instruction of the original language. The result of this activity will be the execution of the original instruction.

The host machine's user assembler language itself will be emulated by HIMICS to allow virtual memory capabilities. HIMICS will also allow "high-level" Languages to be emulated. Languages such as PL/1, APL, and COBOL are likely candidates for emulation. These languages will require a large amount of space for their emulators, but due to the virtual storage capabilities of the system, this space requirement does not present a problem.

1.4 INTER-EMULATOR COMMUNICATION

In many programming situations, it is desirable to use different languages for different modules of the program. For example, one might want the processing portion of his algorithm to be coded in an assembler language, and the input/output sections written in a high-level language. Such process linkage will be allowed in this system. Interprocess communication will also be allowed in this system due to the capabilities of the host machine's operating system. A task or emulated process can start another task executing. After starting another task, the calling task may wait until the named task terminates. The calling task on the other hand may also continue processing and test for the called task's completion when necessary. A task can also cancel another task which is executing. This kind of communication is solely dependent upon the functions of the host operating system.

1.5 MANAGEABLE SOFTWARE

The key to manageable software is to keep it simple. This can be accomplished by using a structured design (3). In this approach a complex system is divided into small independent modules. This allows one to comprehend each module without keeping the details of the entire system in mind. Furthermore, modifications to the system are simplified since a module can be changed or added without affecting other modules.

A part of this structured design is provided by the operating systems of the host processors. Even if the operating systems have to be modified to run in a virtual memory paging environment, it is worth the trade-off. These modules are not only structured, they will be almost error free from the start, and thus more manageable. Obviously it would require a great deal of time to produce the equivalent modules from scratch.

1.6 <u>INSTRUMENTATION</u>

1.6.1 RECORDED COUNTS

In order to evaluate a system's efficiency it is necessary to employ techniques to record the specific actions taken by the system. An obvious method of instrumentation is to keep a count of how many times a pre-specified event occurs. By knowing the system input, and the actions caused, it is possible to evaluate the system.

There are two counts which must be taken for every instruction. The first is a count for each unique operation code. When evaluating the system, the frequency of execution of each kind of instruction is essential. The second count records the number of times each page is referenced. This will be used to evaluate the performance of the system under different paging options.

1.6.2 WORKING SET OPTIONS

Built into the system is the ability to perform paging under one of three options set at system generation time. These three options have been choosen to yield different working set (1,8,9) sizes in order to evaluate the system under various work loads for maximum efficiency. In this paper, working set size refers to the number of pages contained in Level 1 and Level 2 memory. This composes a collection of the program's most recently used pages (11). The terms Level 1 memory and Level 2 memory will be explained in detail in section 1.8.2 of this chapter. The first option operates under a non-competitive fixed partitioned working set size. The second option will allow the system to operate with a competitive variable working set size based on a local paging rate. The third option, which operates on a competitive variable working set size also, is based on a global scale and allocates secondary memory using the LRU stack (1,10) principle.

Under option one, the total amount of secondary memory will be divided by the total number of jobs allowed to be multiprogrammed. This will be set at system generation time. Each job will then have a fixed working set the same size as any other job. For example, if there are 180 page frames and 3 jobs, each job will have 60 page frames for its use. (See Figure 1-3A). Under option two the size of the working set for a job will be adjusted to approach maximum efficiency as best as can be determined. This will be based on a competitive paging ratio computed on a per-job paging rate over a given time period. Jobs having high paging rates tend to increase their working set size while low paging jobs decrease their working set size.

Option three will take the entire working set available and let all jobs have the space needed on a first come first use basis. This treats the Level 2 memory on a global basis, whereas under option one it was treated on a per-job basis or local level. This will allow, for example, two jobs, job 1 needing 20 page frames and job 2 needing 120 page frames, to be completely contained in Level 2 memory at once. (See Figure 1-3B). Under option one above, the fixed size per job, job 2 could only have 60 of its 120 pages in Level 2 memory at once.

Obviously in order to tell which of the above three methods is the best, monitoring of the different options is necessary. This will include a count of the number of page faults occurring out of each memory level for each job under the given option. There will also be an option to turn monitoring on or off.

1.7 OVERVIEW OF THE SYSTEM

The HIMICS system may be viewed as a hierarchical structure (4,11). A structure of this nature consists of modules located

Fixed Working Set Options

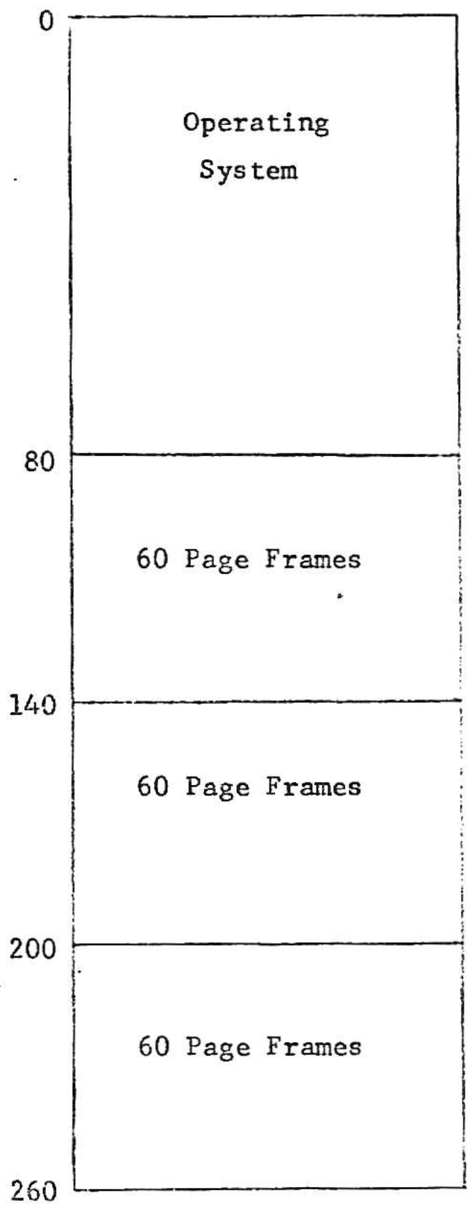


Figure 1-3A

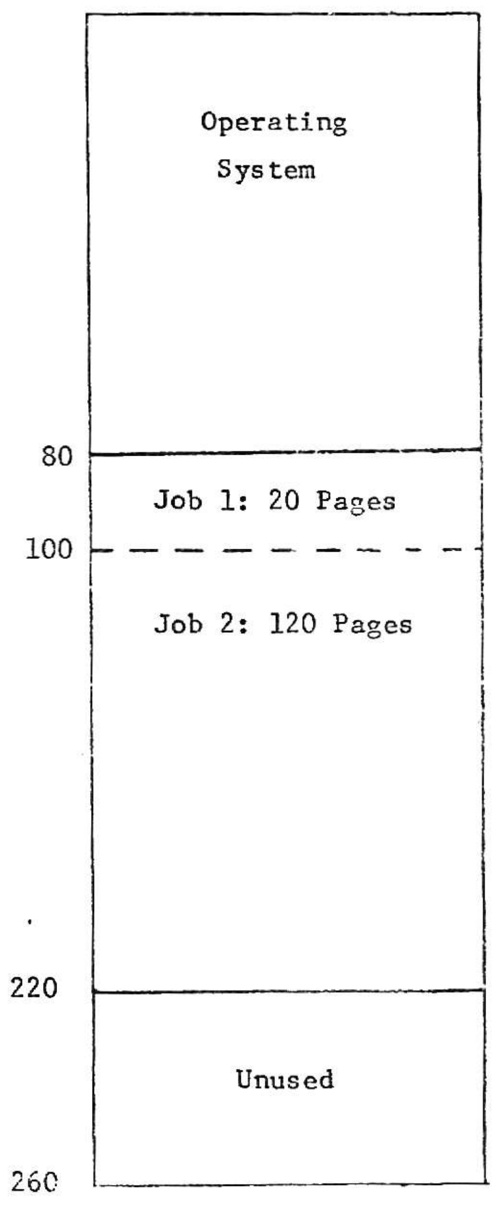


Figure 1-3B

Figure 1-3

on differert hierarchical levels (See Figure 1-4). This type of structure is called "layered insensitivity" (4,11). The levels are insensitive because each level is allowed to call upon the services of levels immediately above or below it in the structure, but not those levels farther than one level away. This means that each level is not concerned about how or where things are done in the levels above it or subordinate to it, and treats them all as one level. Each level may be referenced by the level above or below it in the hierarchy, but no level maybe dependent on a level which is not a logically sequential level in the hierarchy structure. For example, level 4 of the HIMICS system will interface with the file management system level 5, and virtual storage management level 3, but level 4 may not call upon levels 1, 2, or 6.

1.7.1 ADVANTAGES OF SYSTEM DESIGN

There are several advantages to this type of design.

(1) The system is easier to understand.

(2) Each module is easier to implement.

(3) The verification of the entire system is accomplished by verifying each individual level in a bottom-up fashion.

(4) Modification of the system is simplified.

(5) The software system is relatively portable (i.e. interfacing with different hardware requires only the lowest level of the system to be compatible, and the upper levels do not require modification).

Hierarchical Structure

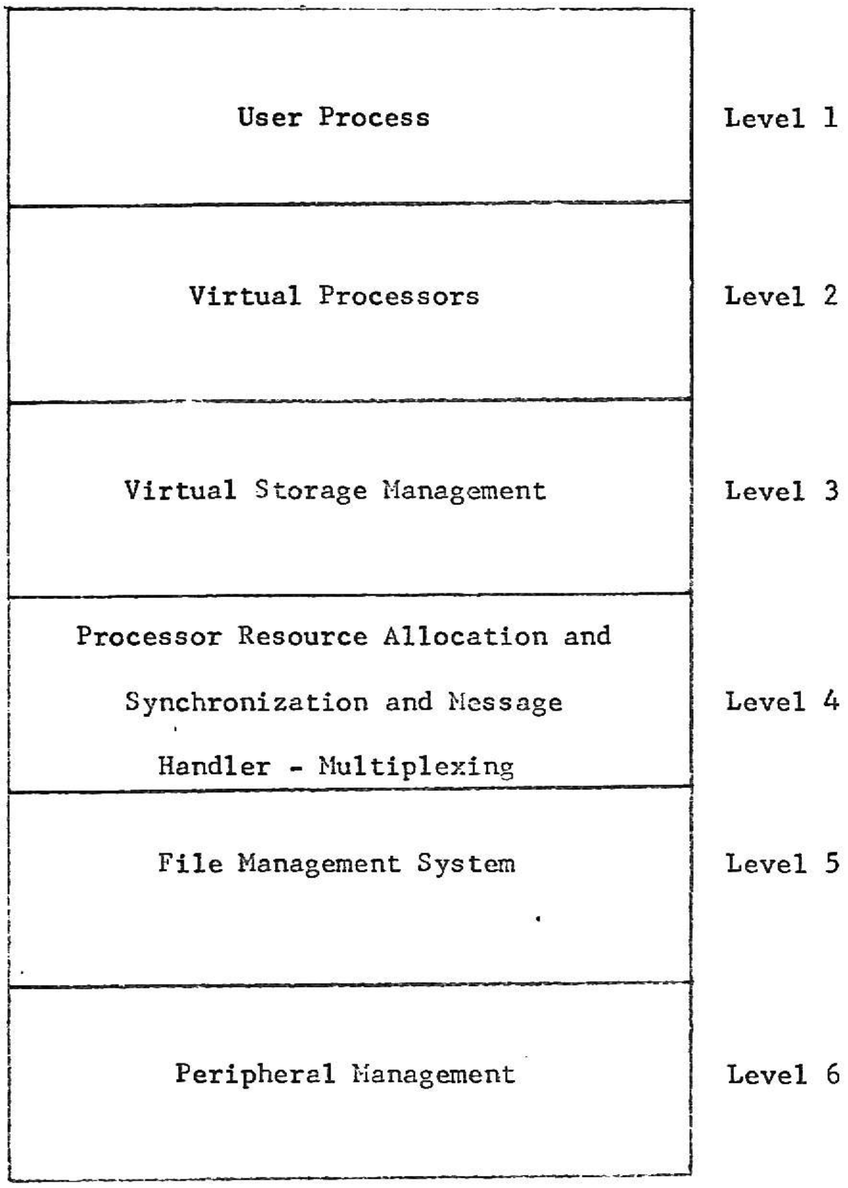


Figure 1-4

1.7.2 EXPLANATION OF LEVELS

A short explanation of what is contained in each module follows. Level 1 contains the user processes which are interpreted and executed in the primary memory of the host machine. These user programs may be written in any language supported by an emulator on the HIMICS system.

Contained in Level 2 are the virtual processors. This is the system of emulators which execute one instruction of the user's program at a time. Before each instruction is executed the current real address of the virtually addressed operands must be retrieved. Therefore all memory references must be detected and sent to level 3 to be converted before the emulation of the instruction may occur.

Level 3 contains the virtual storage management system. This system must detect any page faults which are generated from the virtual addresses of the instruction's operands. The virtual addresses of the operands must be converted to real machine addresses. This of course requires the page to be located in primary memory. This management system must interact with the file management system through the message handler in level 4 to retrieve pages to primary memory.

The message handler and resource allocation systems in level 4 are intermeshed deeply with the operating system of its host machine. The message handler is responsible for the generation and control of all information transfer from the file management system. This communication will consist of I/O messages to and from level 6, page requests from secondary memory in level 5, and the current state of the system (i.e. "request page", "page being transferred"). This message exchange coordinates the activities of levels 5 and 6 with the upper 4 levels of the system.

The file management system will reside at level 5. It will handle all page requests between secondary and primary storage. All input and output messages will be processed by the file management system upon request. Included in the file management module will be tables which contain the current location of each job's pages. These tables are initialized when the job is started, and are updated as pages are moved from one level of memory to another. Level 4 will send page requests and I/O messages through the message handler requesting pages to satisfy page faults and I/O requests when needed.

Level 6 is the peripheral management module. This module will handle the interface with all peripheral devices connected to the system. This may include printers, card readers, teletypes, display terminals, or whatever hardware is available to interface with the system.

1.8 IMPLEMENTATION

We have just presented a machine independent description of the virtual addressing system for a mini-computer system. Now a more detailed description of the implementation of the system at Kansas State University will be presented.

1.8.1 HARDWARE ALLOCATION

The layered insensitivity graph in Figure 1-5 makes the allocation of duties to actual hardware transparent. As can be seen, the upper four modules will be located in an Interdata 85 computer. The lower two levels will be located in a Nova computer.

The upper four levels are located in the Interdata machine because of its greater processing speeds. The extensive software required by the HIMICS system for each user instruction requires a fast processor. The Interdata cycle time of 270 nanoseconds meets these general speed requirements. The Nova machine is used as a peripheral processor. This processor will have more time to perform its duties, and yet removes a great processing overhead from the Interdata. This setup lets each machine do what it does best, and allows a more efficient and faster system. Using two CPU's in effect allows parallel processing. Real I/O may be supervised by the Nova while the Interdata is processing a user's program. Another view of the system design is given in Figure 1-6. The system shown is based on a multiprogramming environment of three users.

1.8.2 MEMORY LEVELS

It is apparent from Figure 1-6 that the use of two separate CPU's primary memory, and disk memory creates three levels of

Hardware View of Hierarchical Structure

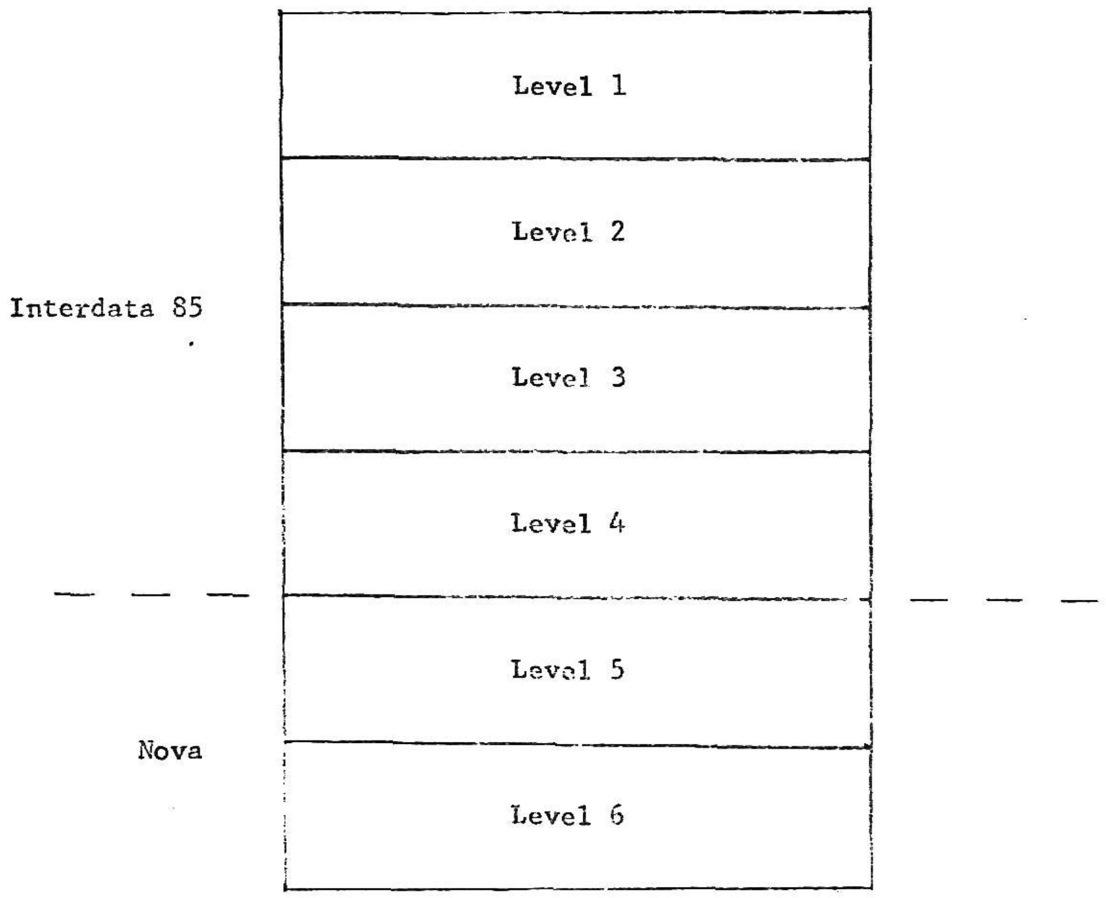


Figure 1-5

Alternate View of System Levels

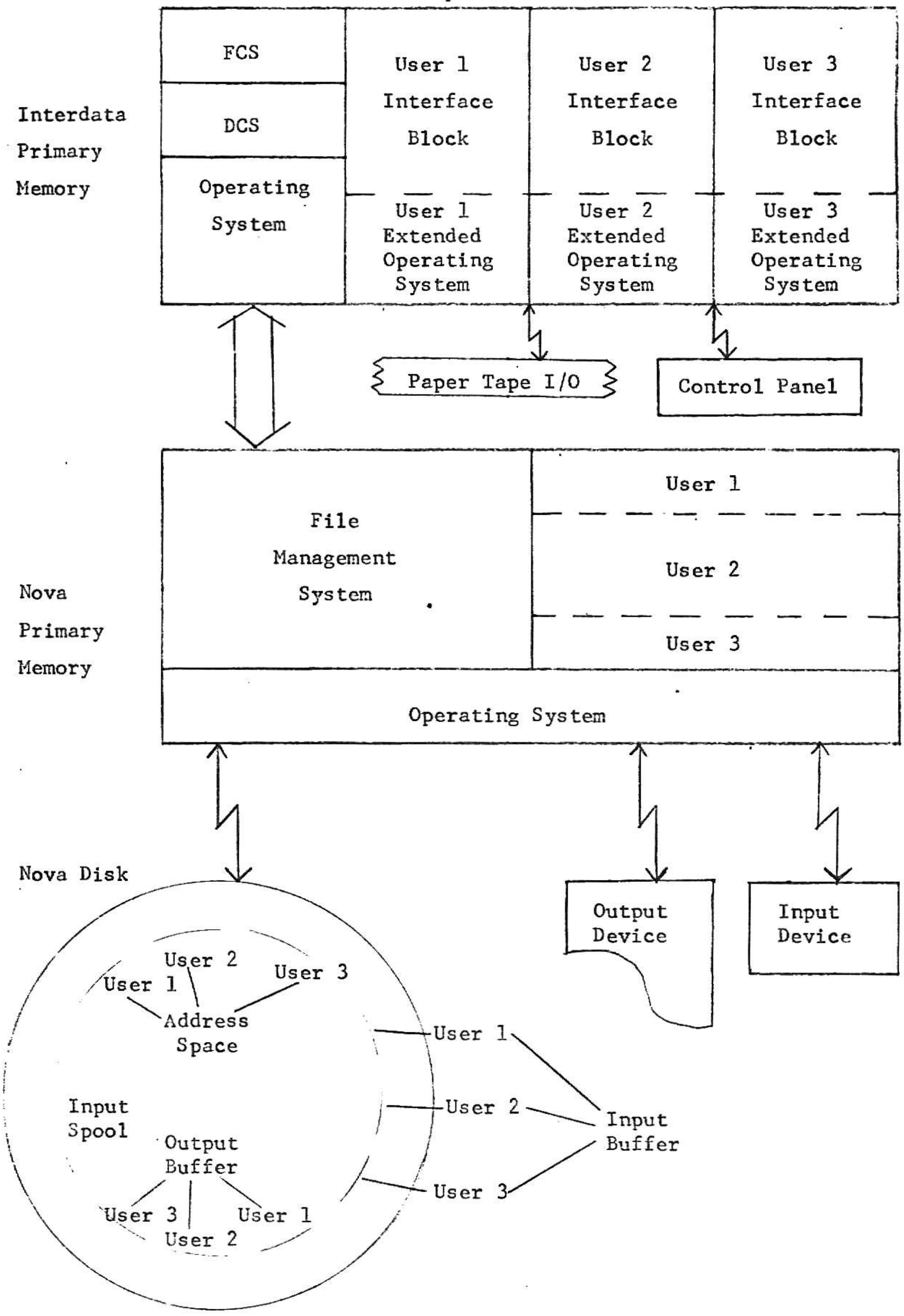


Figure 1-6

memory. These three levels will be referred to as Level 1, Level 2 and Level 3 memory in the remainder of this paper (see Figure 1-7). The Nova disk is the lowest level of memory or Level 3 memory. Each user's program upon entry to the system is spooled to an input file in this third level of memory by the Nova. When the user's job is set to running, the program is put into his address space, also located on Nova disk, Level 3 memory. The data input is stored in his input file. Any output generated by his program will be spooled onto his output file for printing when his job terminates.

Nova primary is the second level of memory for this system. The management system which controls all page traffic in the Nova is located here. This file system receives page traffic from the Interdata, and using its own paging algorithm, rearranges the user's pages in the extended page space in the Nova's Level 2 memory. If a page is being transferred to Level 3 memory (i.e. paged out of Level 2 memory), it is copied back to the address space only if it is an original page. If a page in Level 2 memory is requested by the Interdata to satisfy a page fault, a bit is kept to record whether or not this page is original to the address space in Level 3 memory. If it is, then the page must be copied back to the user's address space before being overwritten when paged out of the Nova's Level 2 memory.

Levels of Memory In The System

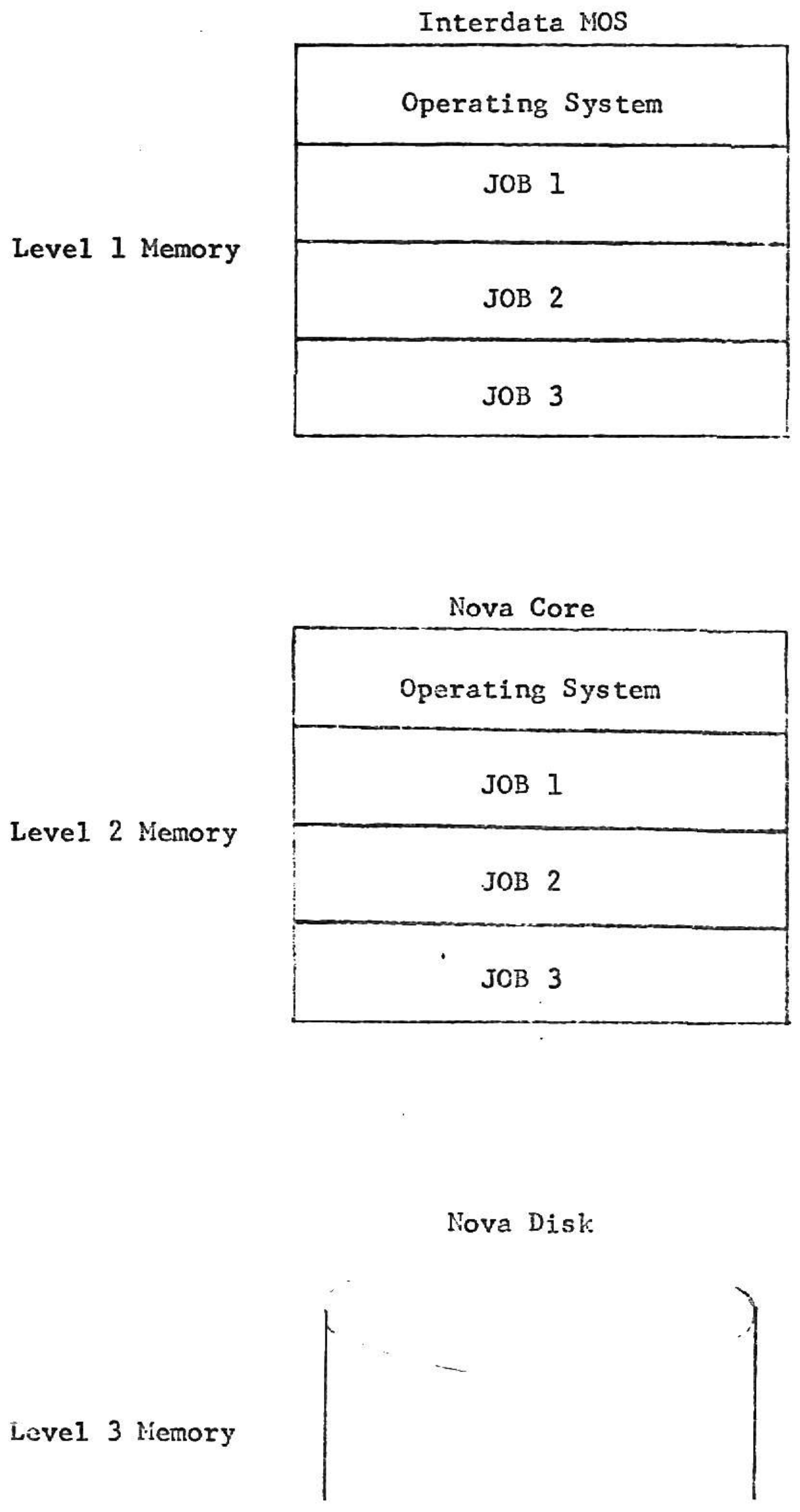


Figure 1-7

This file management system must have tables which keep track of the pages located in its Level 2 memory and Interdata Level 1 memory and all of the user's files located in Level 3 memory. This management system must also interact with the Interdata. All I/O communication and messages must be received and handled by this system.

The first level of memory is located in the Interdata. Located in Level 1 memory will be the user's PCB (Process Control Block). Each user will have space for a fixed number of pages of his program, along with a page table containing information necessary to handle page faults and address mapping.

1.8.3 LOCATION OF SOFTWARE

I/O SVC parameter blocks are created in the user's extended operating system in Level 1 memory. These parameter blocks are needed to inform the peripheral processor of the type of I/O which is to be done. Also the task identification, virtual page number, interval timer and starting and ending location must be included in the parameter block. These parameter blocks are built and then passed to the Nova system.

The FCS (fixed control store) is read only memory which contains the Interdata machine language instruction interpreter. The I/O instructions will not be interpreted by an emulator as are other instructions, but will be interpreted by the program located in the FCS. The dynamic control store (DCS) will contain the language emulator. The address translator, which recognizes and handles page faults and virtual to real address translations, will also be stored in the DCS portion of the Interdata Level 1 memory. The proposed design may be likened to that of a single system. Level 1 Interdata memory corresponds to primary memory. Level 2 Nova core memory, and Level 3 Nova disk memory corresponds to secondary memory.

1.9 SUMMARY

In order to bring the overall picture of the HIMICS system into focus a short summary of the system will be given. This system will be in a multiprogramming environment. Each user will initially have 64K of virtual address space at his disposal. All real I/O will occur in the Nova. A user's source program will be spooled into an input buffer on the Nova's disk. The source program will then be copied onto the user's virtual address space also located on the Nova disk. The user's data will be put in his input file on the Nova disk. The user's source program will be divided into fixed length pages. Each user's program will be given a starting virtual address of zero. As soon as the user's job is set to running by the Interdata operating system, a PCB is created in the Interdata memory. The page table will be located in this PCB.

This table will be empty when the job becomes running. A page table is also set up in the Nova Level 2 memory.

Immediately after a job is started, a page fault will be generated in the Interdata. Page zero will be requested, and paged in from Level 3 memory to Level 1 memory. The system is now ready to begin execution of the program.

The emulator is given the current instruction to process. All operands must be converted to real addresses. When the page needed is not located in the Interdata, a page fault is generated and processed. When all operands are mapped, the instruction may be interpreted. The instruction counter is incremented, and the next instruction is executed (unless the previous instruction was a jump of some kind).

A special case is encountered when the end of job is reached. An I/O SVC must be generated to the Nova to empty the output buffer to the output device. Then a job termination message will cause the address space, buffers, and Nova core page space to be released. The Interdata extended operating system then terminates the job in the Interdata, and a new user's job is initiated.

1.10 INTRODUCTORY DESCRIPTION OF REMAINING CHAPTERS

The remainder of this paper will be concerned with levels one through four of the hierarchical structure, which are the levels contained in the Interdata (Figure 1-5). Levels five and six are included in a similar type report (5).

Chapter two will include a discussion of the crucial algorithms and the data structures associated with them. Chapter three will dwell on the implementation of the system (i.e. algorithms, flowcharts, and software utilization). Finally, chapter four will be a short summary of the HIMICS system and a concluding survey of possible future work which could be done on the system.

CHAPTER TWO

2.1 INTRODUCTION

We present in chapter two of this paper a discussion of page replacement, address mapping algorithms, and the associated data structures. The Interdata emulator and message exchange between processors will also be described. Finally a trace following a program through its execution in the system will be given. This example will demonstrate how the entire system interacts and how the module communicate with each other.

2.2 PAGE REPLACEMENT

Page replacement strategy for the HIMICS system's primary memory will be an implementation of the LRU (Least Recently Used) algorithm. The approximation algorithm to be used is called NUR (Not Used Recently) (1). An LRU algorithm records when each page located in primary memory was last referenced (1). When a page fault occurs, the page which has not been referenced for the longest time is removed to allow space for the requested page. If the page scheduled for removal is needed by the instruction which caused the fault, it must be locked in and another unused page is paged out instead. The LRU algorithm is an excellent paging algorithm, but the continual reference updating is too time consuming when done by software. The LRU approximation algorithm, NUR, uses a reference bit instead of a reference time. Each page has this reference bit associated with it in the page table (See Figure 2-1). All reference bits are

Page Table (PAGE-TAB)

	REF-BIT	ORG-BIT	LOCK-BIT	PRES-BIT	PTR	MONITOR
Page Number	Bit 0	Bit 1	Bit 2	Bit 3	Bits 4-10	Bits 11-31
0						
1						
2						
3						
⋮						
124						
125						
126						
127						

Figure 2-1

periodically set to zero after every page fault. Whenever a page is referenced, its bit is set to one. When a page fault occurs, the first page found with a reference bit of zero is paged out.

Algorithm 2-2 is a NUR page removal algorithm which is employed when a page fault occurs. Before discussing the algorithm, the data structures used will be described.

In order to determine whether or not a page fault has occured, it must be known whether or not the page needed is in Level 1 memory. A bit in the page table will indicate whether or not that page is in Level 1 Interdata memory. A list of the pages which are present in Level 1 memory is used to correlate the start of the page with a real address in Level 1 memory. This table of page numbers in Level 1 memory is called CURRENT-PAGES (Figure 2-3).

The Level 1 memory partitions for each user will be of fixed size in this system, although a variable size partition scheme may be added without a great deal of system modification. The addresses in each fixed partition will be the same for different jobs (i.e. the starting page addresses inside each fixed partition must be known for virtual to real address conversion). Each user partition will have a table of the starting page address within that partition. The values in this table, called PAGE-ADDR (Figure 2-4), will not change. The number of

Page Fault Handler (FAULT)

```
FAULT:   BEGIN;
              PAGE-TAB.REF-BIT(REQ-PAGE)=1;
              PAGE-TAB.PRES-BIT(REQ-PAGE)=1;
              PAGE-TAB.PTR(REQ-PAGE)=REM-PTR;
              CURRENT-PAGES(REM-PTR)=REQ-PAGE;
              CALL READ-SVC(REQ-PAGE,PAGE-ADDR(REM-PTR));
              DO WHILE('1'B);
              REM-PTR=REM-PTR+1;
              IF REM-PTR=MAX+1 THEN REM-PTR=1;
              IF PAGE-TAB.REF-BIT(CURRENT-PAGES(REM-PTR))=0 &
              PAGE-TAB.LOCK-BIT(CURRENT-PAGES(REM-PTR))=0 THEN DO;
                   CALL SEND-SVC(CURRENT-PAGES(REM-PTR),PAGE-ADDR(REM-PTR));
                   PAGE-TAB.PRES-BIT(CURRENT-PAGES(REM-PTR))=0;
                   CURRENT-PAGES(REM-PTR)=NULL;
                   PAGE-TAB.REF-BIT=0;
                   RETURN;
                   END;
              PAGE-TAB.REF-BIT(CURRENT-PAGES(REM-PTR))=0;
              END;
         END;
```

Algorithm 2-2 (continued on the next page)

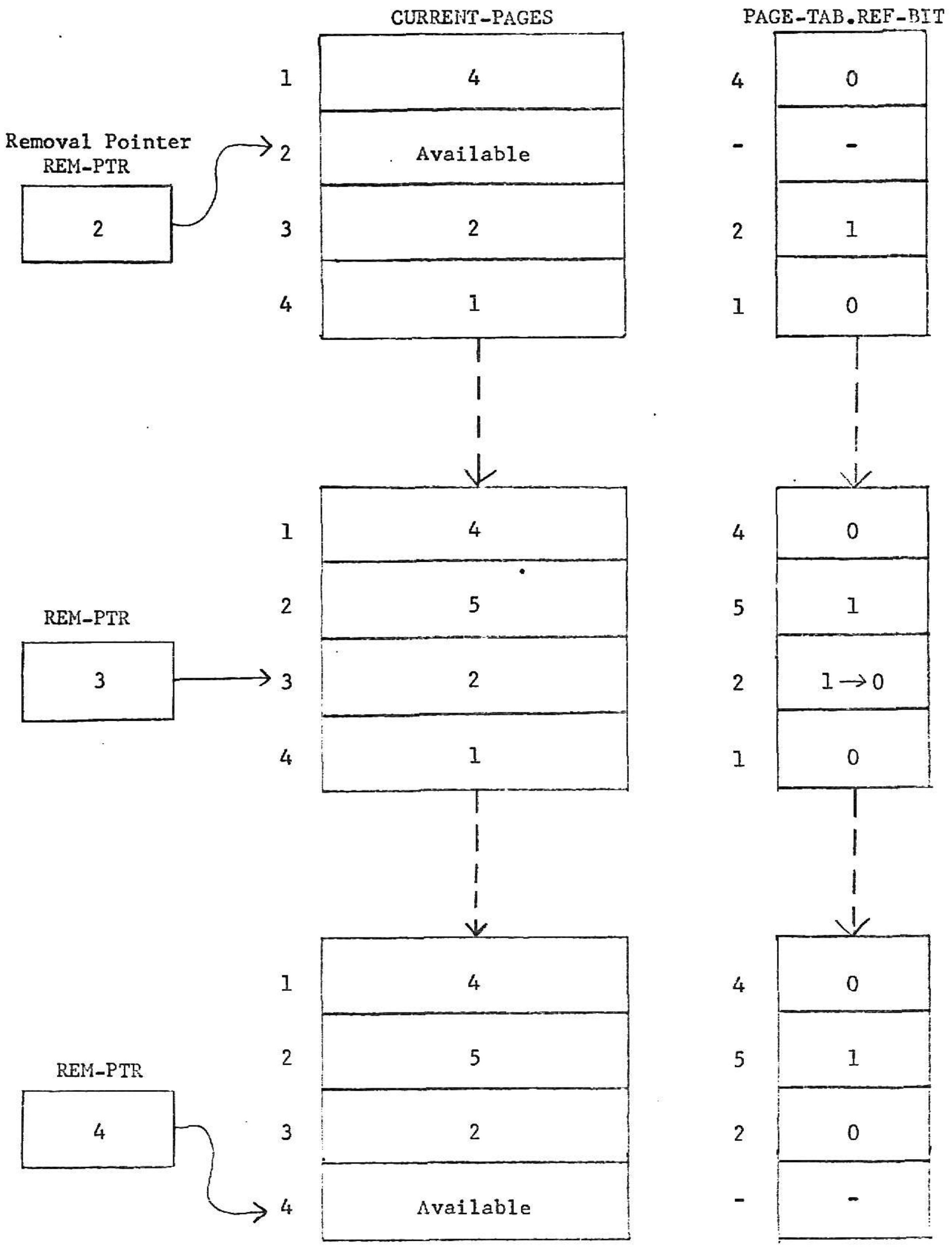
Term Definitions

PAGE-TAB...........	The page table (figure 2-1).
REF-BIT............	Bit zero in the page table (figure 2-1).
REQ-PAGE...........	The page needed by the user program which caused the page fault. For figure 2-3 it is page 5.
REM-PTR............	The pointer to the table CURRENT-PAGES, which always points to the available space in the table.
ADDR-PTR...........	Holds the value of REM-PTR.
MAX................	The number of pages allowed in primary memory at one time.
CURRENT-PAGES.......	A table which contains the page numbers of the pages which are currently located in primary memory.
NULL................	Blanks.
LOCK-BIT............	Bit two in the page table (figure 2-1).
SVC.................	Symbolizes the transmission of a page request to the Nova. The page required, and the recieving address are passed as parameters.

Algorithm 2-2

(continued from preceding page)

The Structure CURRENT-PAGES



The structure CURRENT-PAGES during the execution of the NUR algorithm. The reference bit of page table is also shown.

Figure 2-3

The Structure PAGE-ADDR

0	8192
1	8704
2	9216
3	9728
4	10240
5	10752
6	11264
7	11776
8	12288
9	12800
10	13312
11	13824

Figure 2-4

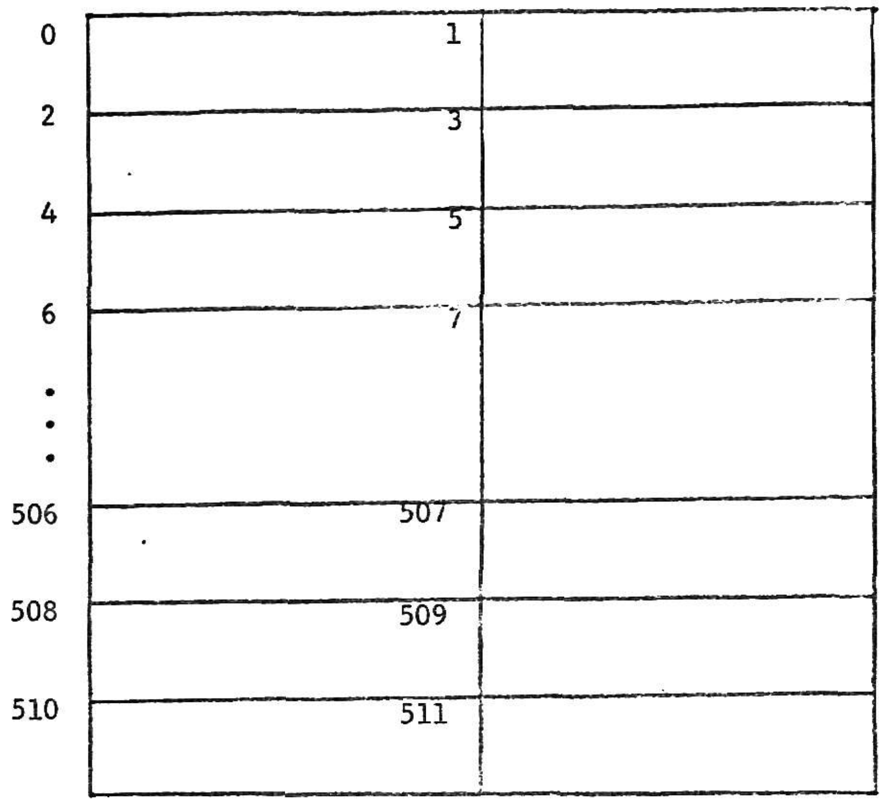
page spaces in each partition will be one greater than the maximum number actually located there. This will allow processor overlap during the swapping of pages. The Interdata may work on the NUR paging algorithm after requesting a page from the Nova. The Nova will be processing the page fault and transferring the requested page at the same time the Interdata is sending the page which was removed from the user's partition. The page space which is free will be a different one each time since the new page is put in the available space, and another page is removed which creates the new available page space. A pointer called REM-PTR will index the available page space address in PAGE-ADDR. The page number of the page which is located at the address in the first element of PAGE-ADDR for example, will be in the first element of CURRENT-PAGES. In this way, the Interdata address for each page will be known.

When a page fault occurs, the proper message requesting the needed page is sent to the Nova system. The Level 1 memory location given to receive that page is in PAGE-ADDR(REM-PTR). The page number of the requested page is put in CURRENT-PAGES(REM-PTR). The reference bit and present bit are set to one in PAGE-TAB. The pages in CURRENT-PAGES are looked at sequentially from element REM-PTR+1 until a page with a reference bit of zero and a lock bit of zero is found. This is the page which will be sent back to Level 2 memory. The reference bits of the pages in CURRENT-PAGES are set to zero during the removal search. This is in case all reference bits are one to start with and the search goes through the table without finding a page to remove. After a page has been found, the reference bit of all the pages is set to zero. REM-PTR now points to the empty page space address in the partition. The corresponding element of CURRENT-PAGES is set to null. The free memory space is not destroyed until the next page fault when it will be overwritten by the incoming requested page.

2.3 ADDRESS MAPPING

The process of mapping a virtual address to the actual machine address is a simple one. The page table for this system will allow 128 entries (Figure 2-1). Each page will contain 256 words. Every byte (8 bits) on the Interdata is addressable. This means each page will contain 512 addressable bytes that may be referenced (Figure 2-5). When a virtual address of 16 bits is given, the left seven bits will define the page number, and the right nine bits will be the offset from the start of the page (Figure 2-6). The page number determined from the virtual address is used as an index in PAGE-TAB. The present bit for that entry is tested. If the page is not currently in primary memory, a page fault is generated. If the page is present, then the pointer to PAGE-ADDR is taken from PTR in the page table. The contents of element PTR of PAGE-ADDR is taken as the starting Interdata Level 1 memory address for this page. The offset determined from the virtual

Sample Page (page zero)



Each byte of a word is addressable. Thus there are 512 adresses on a 256 16-bit word page.

Figure 2-5

Example Virtual Address (VIRT-ADDR)

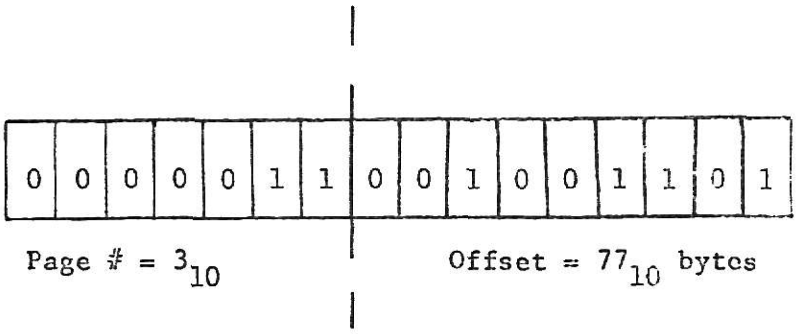


Figure 2-6

address is added to the starting address of the page to form the effective real address of the operand. The algorithm which checks for a page fault, and eventually performs the address map, is given in Algorithm 2-7.

2.4 INTERDATA EMULATOR

The reasons for emulating the host machine's assembler language are threefold. First of all, in order to implement a virtual addressing system completely by software with no hardware support, it is necessary to capture each memory reference. Page faults and address maps must be processed before the instruction may be executed. The second reason for emulation is that all I/O instructions must be processed by the extended operating system and not by the altered Interdata interpreter. The reason for this special handling is because all peripheral management and I/O is performed in the Nova machine (See Chapter 1). The third and final reason for emulation is that privileged instructions will not be processed by the Interdata emulator. Privileged insructions will not be allowed in the HIMICS system at all.

The logic of the emulator will be like that of a control process which transfers to different routines to accomplish the needed tasks. First a count of instructions types (i.e. how many of each type instruction is executed) may be kept. This is for monitoring purposes. When this monitoring is in effect, a counter for each type of instruction is kept and incremented

Algorithm 2-7 Address Mapper

```
ADDRMAP: BEGIN;
            REQ-PAGE=VIRT-ADDR(bits 0-6);
            OFFSET=VIRT-ADDR(bits 7-15);
            IF PAGE-TAB.PRES-BIT(REQ-PAGE)=0 THEN DO;
                CALL FAULT(REQ-PAGE);
                ADDR-PTR=REM-PTR;
                END;
            ELSE ADDR-PTR=PAGE-TAB.PTR(REQ-PAGE);
            REAL-ADDR=PAGE-ADDR(ADDR-PTR)+OFFSET;
        END ADDRMAP;
```

ADDR-PTR............	Pointer to the element of CURRENT-PAGES which contains REQ-PAGE.
CURRENT-PAGES.......	Vector containing the page numbers of the pages which are currently in Level 1 memory.
FAULT...............	Algorithm 2-2.
OFFSET..............	Right nine bits of VIRT-ADDR.
PAGE-ADDR...........	Figure 2-4. Locations of pages in Level 1 memory.
PRES-BIT............	Bit three of the page table.
PTR.................	Bits four through ten of the page table.
REAL-ADDR...........	Computed operand address in Level 1 memory.
REQ-PAGE............	Left seven bits of VIRT-ADDR.
VIRT-ADDR...........	Operand address in user program.

Algorithm 2-7

every time that type of instruction is found. When the current instruction is an I/O instruction, then the virtual addresses are mapped and page faults are processed. Control is then transferred to the extended operating system for execution. If the current instruction is not an I/O instruction, a privileged instruction is looked for. When it is a privileged instruction, the appropriate error message is generated and the process goes to end of job.

When any other type of instruction is encountered, similar action is taken. The instruction's counter is incremented (if the monitor option is in effect), and a microcode routine is called which executes the instruction. After the execution of the instruction, the instruction pointer is adjusted, and the next instruction is processed.

2.5 MESSAGE EXCHANGE

In order for this system to function properly, there must be considerable communication between the two CPU's. In the case of an I/O communication, a parameter block defining the operation to be performed is passed to the Nova from the Interdata. This information includes an I/O code to define the type of I/O. The task which is doing the I/O must also be identified. The locations of the data involved must also be included in an I/O communication. This parameter block is described in greater detail in chapter 3.

Of course the state of the system (i.e. operation completed etc.) must also be passed back and forth. Communication concerning page faults will also be passed, although the actual transfer of data will occur through a direct channel from the Nova to the Interdata. The message exchange will be the method of coordinating the activities of the two computer systems to operate as one system.

2.6 EXAMPLE OF A TASK EXECUTION

In the multiprogramming environment of this system, each user will have 64K bytes of virtual memory space at his disposal. The user's program will enter the system in the Nova. The source program will be spooled into an input buffer located on the Nova disk space. The source program is then copied into the user's address space where it will be divided up into fixed page segments of 256 words each. The length of 256 words is due to the nature of the construction of the disk. The input data will be put in the user's input file.

The virtual environment will allow each user's program to be given a starting virtual address of zero. That is, page zero contains virtual addresses 0-511. When the job is started by the Interdata operating system, a process control block (PCB) is created in Interdata primary memory. Assuming X is the maximum number of pages located in primary memory at any one time, memory space for X+1 pages of the user's program will be kept in the user's PCB. The page table, as stated earlier may have up to 128 entries. Each page entry will contain a reference bit, an original bit, a lock bit, a present bit, a seven bit pointer, and a monitor counter to count the references to each page (Figure 2-1). The tables CURRENT-PAGES (Figure 2-2), and PAGE-ADDR (Figure 2-4) along with the page table are all located in the tasks' PCB. All data structures are empty when the job is initiated. The file management system in the Nova will keep track of the user's space in that part of the system (5).

When set to running by the operating system for the first time, a page fault will be generated, because the page containing the first instruction to be executed is not in Level 1 memory. Page zero will be requested by the Interdata system, and the Nova system will load the page into primary memory through a direct channel. An instruction pointer in the task's PCB will keep track of the current instruction to be executed. The page containing that instruction must be in the Interdata before it could possibly be executed by the emulator.

When an I/O instruction is detected, the pages involved are determined. The pages located in the Interdata will be transferred back to Level 2 memory on request if the original bit shows that the page has been altered (i.e. if the original bit of the page table is one). If the page is not original, it will not be transferred. If the I/O is a read then the page will be purged if it is not the upper or lower boundary page. The I/O instruction is now processed as explained earlier.

The other instructions will be handled as explained earlier in the emulator description. Each memory operand if any, is given to the virtual address translator which searches the CURRENT-PAGES table to determine whether or not the page to be referenced is in Interdata memory (Algorithm 2-7). A page fault is generated when the page is not present. The page fault routine eventually requests a certain page be transferred to primary memory. A virtual page number along with the available address from PAGE-ADDR are passed as parameters. The Nova locates the desired page (5) and retrieves it from Level 2 Nova memory or from Level 3 memory (the address space on disk), and through the data channel copies it into Interdata memory. The page to be removed from the Interdata must always be recopied back to the Nova system.

Once the page fault has been processed, the virtual address for the operand(s) must be converted to an Interdata address. The address mapper uses the table PAGE-ADDR in the PCB to calculate this address. The procedure is described in Algorithm 2-7. After the address translation is complete for the first operand, the same procedure will be followed for each memory reference in the instruction.

The instruction is now executed by the emulator and the interpreter routines in the DCS. During execution, the instruction pointer is altered (incremented unless a branch occurs), and points to the next instruction to be executed.

When all the instructions have been executed, and the end of job situation arises, all of the original pages in the user's partition must be sent back to Level 2 memory. A message is sent to the Nova to output the user's buffer. While the Nova is dumping the output buffers, the Interdata may be reinitializing the partition for the next user process. When the Nova sends a completion message back to the Interdata, a job termination message is sent to the Interdata extended operating system. The job is then terminated, and another one from the ready list is started. This process is discussed in greater detail in chapter three.

CHAPTER THREE

3.1 INTRODUCTION

This chapter will be devoted to describing critical areas of implementation of the HIMICS system. The host operating system in the Interdata, OS/16 multi-tasking (OS/16-MT), will be used to achieve many necessary functions of the HIMICS system. A general description of OS/16-MT is therefore presented. Also in chapter 3, the user's extended operating system will be described. Its functions will be given and the general algorithm of its logic will appear in the appendix. The ISP(13) of the Interdata 85 emulator will appear in the appendix also and will be described in this chapter. A comparison will be made with the microcode equivalent to the ISP for two sample instructions. Finally, a system control, high-level flowchart which appears in the appendix will be discussed.

3.2 OS/16-MT

This multi-tasking operating system (14) performs system control and scheduling functions. The number of tasks is variable under OS/16-MT, but will be fixed for the HIMICS system each time the system is generated. A task may consist of a single program, or it may be a main program with one or more subprograms. There are two types of tasks: system tasks, and user tasks. Each user task will be given a priorty of 0-15 with zero being the highest priority.

3.2.1 SYSTEM STATES

A task may exist in one of eight states under OS/16-MT:

(1) Dormant

(2) Active

(3) Ready

(4) Task wait

(5) Console wait

(6) I/O wait

(7) Time wait

(8) Overlay wait

Dormant means the task has not started, or has gone to end of job. In the HIMICS system, a task will not exist in the system after it has terminated. A completed task will be purged and a new task brought in; therefore, all dormant tasks will be those waiting to be started for the first time. An active task is the one currently executing instructions. Only one task exists in this state at one particular time. When a task is in the ready state, it will start or resume execution when it becomes the highest priority ready task. The task wait state indicates that the task has called another task into execution and is waiting for the called task to go to end of job. Console wait means the task is waiting for a reply from the console. I/O wait means the task is waiting for completion of an I/O request. Time wait means the task is waiting for a specific interval to elapse, or for a specific time of day. Overlay wait means the task is waiting for an overlay to be loaded.

3.2.2 SYSTEM GENERATION

OS/16-MT is a memory resident system. When the system is generated, the assembled user tasks are loaded with the operating system modules. In a normal situation, all the user tasks to be executed would be loaded at this time. They would then be run to completion, and the system would terminate. In order to run more user tasks, the system would have to be generated again. Under the HIMICS system, OS/16-MT will stay operative while new user tasks are brought in when other user tasks reach end of job. The end of job instruction will not be executed in the Interdata. Instead, a message will be built in the extended operating system and sent to the Nova system to bring in another job. In this manner, a completed task is removed from the system and a new one brought in. OS/16-MT will still think it is the original task. Therefore, if generated with five user tasks, the HIMICS system might process 20 jobs until given a special system termination message from the console.

3.2.3 TASK CONTROL INFORMATION

For each task present at system generation time, a memory area is reserved for control information by OS/16-MT. System-oriented parameters associated with the task are maintained in a

Task Control Block (TCB) separate from the task itself. User oriented parameters are contained in a Task Block (TB) at the beginning of the task. The HIMICS system creates a Process Control Block (PCB) for each user task no matter when the task enters the system. This PCB is created and stored in the extended operating system. OS/16-MT refers to tasks and schedules tasks via their TCBs. Scheduling is performed on a priority basis. Each task has a priority assigned in its TCB. When a task is interrupted, the Program Status Word (PSW) is saved in its TCB and registers are saved in the TB associated with that task. The task ID, status of the task, TB address, initial PSW, priority, current and restart PSW, and a pointer to a TCB table (TCBTAB) are all kept in the TCB for each task. TCBTAB contains an entry for each TCB in the system. The TB contains the register save areas, the boundary addresses for the task, and a parameter word for passing a parameter to a called task. The PSW and registers are restored each time a task is made ready for execution. Parameters from the system or from other tasks may be passed to a task, thus enabling tasks to be designed conveniently to perform functions in a re-entrant manner.

3.2.4 PARAMETER QUEUE

OS/16-MT enables 16 bit parameters, from either the system or other tasks, to be queued with a task. This feature requires the task to allow queuing and the task program to begin with a circular list of any size. When passing a parameter to a task's parameter queue, the operating system puts the task in the ready state if dormant. The parameter may indicate a certain event or may point to a table of parameters that the activated task should process. This queuing feature allows one task or several tasks to queue jobs with another task. When passing the queue parameter to the task's list, the system adds it to the bottom of the list. When servicing the list, the task should remove from the top of the list for a FIFO operation. When the task finds the list empty, it can safely terminate itself. The HIMICS system will use this parameter queuing facility to keep the task ID alive when the end of job has been reached, and a new task may not terminate while there are parameters present in its queue.

3.2.5 SUPERVISOR CALL INSTRUCTION

The supervisor call instruction (SVC) is a powerful tool in the Interdata assembler language. OS/16-MT treats this instruction, under normal operations, as an internal interrupt. The form of the SVC instruction is:

SVC R1,A(X2)

where the R1 field is a value rather than a register. The A(X2) field is an address which points to a parameter block. The R1 value defines the type of SVC while the parameter block determines the precise operation desired and the necessary information to

SVC Instruction List (14)

SVC 1,PARBLK	Input/output operations
SVC 2,PARBLK	Code 1 - pause
	Code 2 - get storage
	Code 3 - release storage
	Code 4 - set status
	Code 5 - fetch pointer
	Code 6 - unpack
	Code 7 - log message
	Code 8 - interrogate clock
	Code 9 - request date
	Code A - time wait
	Code B - interval wait
	Code C - log message and await responce
SVC 2,0	End of job
SVC 5,PARBLK	Fetch overlay
SVC 6,PARBLK	Call task
SVC 10,PARBLK	Cancel task

PARBLK is an address (address plus index register) of a block of storage which is used to define the desired operation.

Figure 3-1

satisfy the requested operation.

In the HIMICS system SVC instructions will be trapped in the emulator and passed to the extended operating system. In the majority of cases, another SVC will be created there to perform the operation. Figure 3-1 shows a summary of OS/16-MT SVC instructions.

3.2.5.1 SVC 1-I/O INSTRUCTIONS

Under OS/16-MT, the user does not issue I/O commands directly; instead, he defines the I/O operation and the operating system does the I/O. At the user level, the SVC 1 will be the same under the HIMICS system. The parameter block for an I/O SVC is shown in Figure 3-2.

User Parameter Block

0 7	8 15
Function Code	LU
Status	Device Address
Starting Address	
Ending Address	
Relative Address	
Write Key	Read Key

Figure 3-2 (14)

The function code is an eight bit field which defines the I/O operation (i.e. write ASCII and wait, read binary and proceed, etc.). The logical unit (LU) is just a device code for the I/O device desired. The status byte will receive a return code after the operation, and the device address byte will receive the physical address of the device. The next two parameters are virtual addresses which define the starting and ending address for the operation. The next two parameter words are optional, and will not be discussed here.

When an I/O SVC is decoded, the emulator transfers control to the extended operating system. There an SVC is created with the user's I/O parameter block together with an additional parameter block created in the extended operating system as data. The two parameter blocks are used as the data which is passed to the Nova system. In this manner, the user's parameter block defining the I/O operation and the system generated task blocks are moved to the Nova system. The Nova system will parse the parameter block and proceed with the I/O operation. This procedure will be different for a read than for a write I/O. When a read operation is specified, all full pages located within the receiving addresses will be deleted from Level 1 Interdata memory. Pages which are only partially involved in the operation (i.e. the beginning page if the beginning address is not on a page boundary, and the ending page if the ending address is not on a page boundary) are locked in if they are located in Level 1 Interdata memory. When a write operation is specified, all pages located in Level 1 Interdata which are involved in the operation are locked in. The Nova must be informed as to which ones are original. No pages are transferred out unless they are asked for by the Nova system.

The contents of the additional parameter block are shown in Figure 3-3.

System Generated Parameter Block

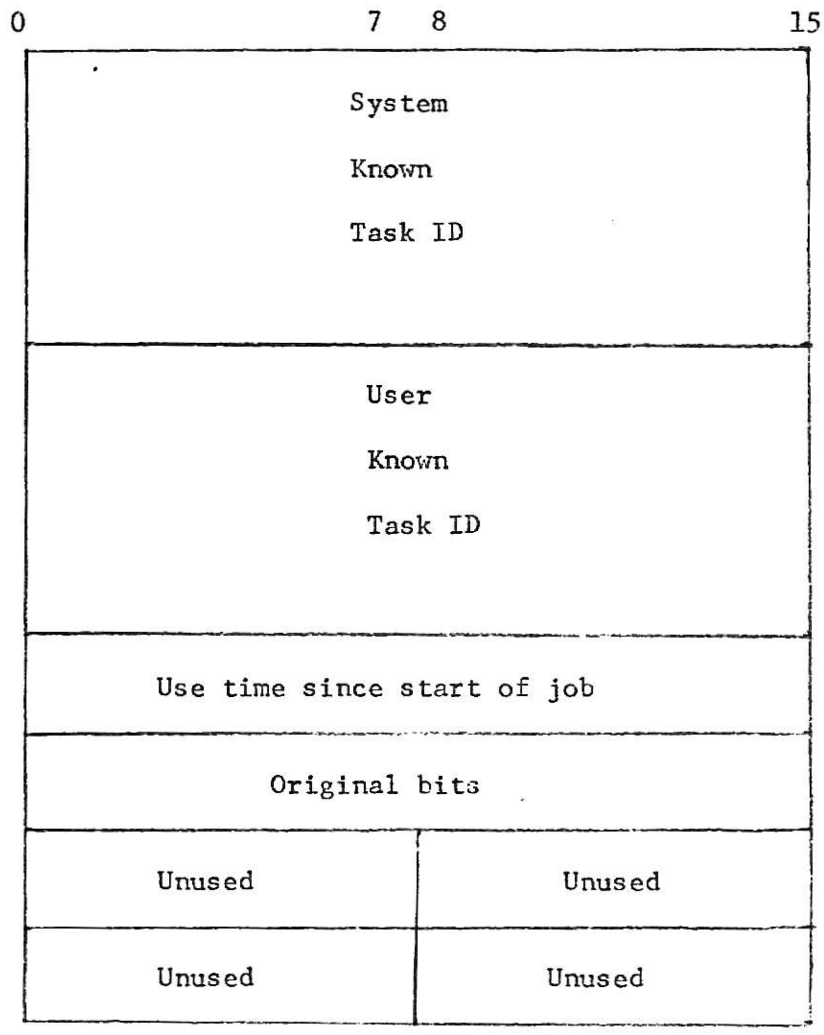


Figure 3-3

These additional parameters will be passed to the Nova system. Both names for the task (the task name known by OS/16-MT at system generation, and the task name of the current task) must be passed to identify to the Nova which task the operation is for. The time passed is task CPU time used to this point. The universal clock time is used to keep track of the task's CPU time in microseconds. The original bits of the pages located in Level 1 memory are passed to the Nova. A specified word of the parameter block will contain a bit string. The high order bits in this string will match up with the order in which the pages appear in the page space for the task in Level 1 Interdata memory.

3.2.5.2 SVC 2 - SERVICE FUNCTIONS

Service functions are obtained by an SVC 2 under OS/16-MT. The function is defined in a parameter block by a function code. These functions will be executed as normal under HIMICS except that they will be invoked from the extended operating system rather than the user's program.

3.2.5.3 SVC 3 - END OF JOB

An SVC 3 instruction is the normal user command to signify the end of his task. In the HIMICS system, this will initiate the end of task clean-up process. The task will be purged from the system, and another task loaded in its place. This procedure maintains the system until there are no more jobs to be run.

3.2.5.4 SVC 5 - FETCH OVERLAY

This SVC instruction will be processed in the extended operating system. A fetch overlay will be treated as a read and wait I/O instruction. The procedure followed is identical to that described in section 3.2.5.1 for a read I/O function. A parameter block must again be passed to the Nova system.

3.2.5.5 SVC 6 - CALL TASK

This SVC permits a task to call another task and pass a parameter. Three types of calls are supported by OS/16-MT.

(1) Start task and proceed.

(2) Start task and wait.

(3) Wait for a called task.

In a start task and proceed, the calling task may continue execution even though the called task may not have finished execution. In a start task and wait, the called task must complete executing before the calling task may continue. In the last type, wait for a called task, the called task is not started, but the calling task must wait until the called task has completed execution before it (the calling task) may continue. A parameter block is used with this SVC call. The task ID is included in this block although it is not the same name the system would recognize. This is because the operating system knows only the original name of the job which was present when the system was generated. The name is treated as a partition name while the actual task ID is located in the PCB for that partition.

If the SVC is a start task and the task is dormant, the parameter word of the parameter block will be queued if specified (section 3.2.4). If the calling task is to proceed, then both tasks are placed in the ready state. Otherwise the called task is put in the ready state, and the calling task is put in the task wait state until the called task goes to end of job. Also the parameter cannot be queued. If the called task is not dormant, the task start will not be accepted except in the case of a start task and proceed with a parameter queue. With a wait for a called task, the called task may not have a parameter queue. If the called task has not yet reached end of job, the calling task is placed in the task wait state until it does.

3.2.5.6 SVC 10 - CANCEL TASK

This SVC may cancel a task, including itself. A cancellation will terminate any I/O being performed by the task at the time it is cancelled. The cancelled task is made dormant, and may be restarted at a later date by the call task SVC (section 3.2.5.5). In the HIMICS system, this will not be possible because once a task has reached end of job, it is purged from the system. If a task is waiting for the cancelled task, the waiting task is made ready. The parameter block associated with this SVC instruction consists of the task name of the task to be cancelled, and a status field which effectively acts as a return code.

3.3 EXTENDED OPERATING SYSTEM

When OS/16-MT and HIMICS is generated, a program and storage block called the Extended Operating System (EXTOS) is created which serves as an interface between the emulator and the host operating system. The EXTOS is set up for each task in the HIMICS multi-tasking environment. The program section of the EXTOS will be coded in Interdata 80 user assembler language. Once the system has been generated, the EXTOS remains with the user page space and Task block while different tasks enter and leave the system (section 3.2.2). The EXTOS program will have 2 distinct sections. A high level algorithm description of the EXTOS programming functions is given in Appendix A. The storage block of EXTOS is described first.

3.3.1 PROCESS CONTROL BLOCK

The Process Control Block (PCB) is a storage area in the EXTOS. The PCB will contain the current user task ID, instruction pointer, the page table (PAGE-TAB), a list of pages currently in Level 1 Interdata memory (CURRENT-PAGES), and a list of the real Interdata addresses for the pages in the user's page space (PAGE-ADDR).

When the system is generated with a fixed number of partitions, each partition will have a fixed task name or identification (ID). The individual user tasks will all have a unique name given by the user. This unique task name will be stored in the PCB space. This must be done in order to allow inter-task communication. When a task ID is given for a start task, the task

ID must be searched for in the PCB of all partitions in the system. The search will be done in the EXTOS routines which also issues the SVC to communicate with the target task.

The instruction pointer for the task will be stored in the PCB. This pointer contains the virtual address of the instruction to be executed. This pointer is either updated during the instruction, or after execution of the current instruction.

The contents of the page table are described in chapter two. This table is used by the HIMICS system to store the lock bit, reference bit, original bit, and reference count for each page of the user's program.

The contents of CURRENT-PAGES is a list of the pages which are currently located in Interdata Level 1 memory. This list is used by the NUR page fault algorithm (Algorithm 2-2).

The contents of PAGE-ADDR is a list of the beginning page addresses in the user's Level 1 page space. These addresses are real Interdata Level 1 memory addresses. The elements of this list correspond element by element with the contents of CURRENT-PAGES.

Figure 3-4 shows the placement of the PCB with the rest of the user's partition. Also shown are the routines which execute user SVC instructions. These routines are described next.

3.3.2 SVC INSTRUCTION EXECUTION

When an I/O SVC is to be executed, the user's parameter block from the user's program is brought into EXTOS. This

EXTOS Block Contents
and
Placement in the Partition

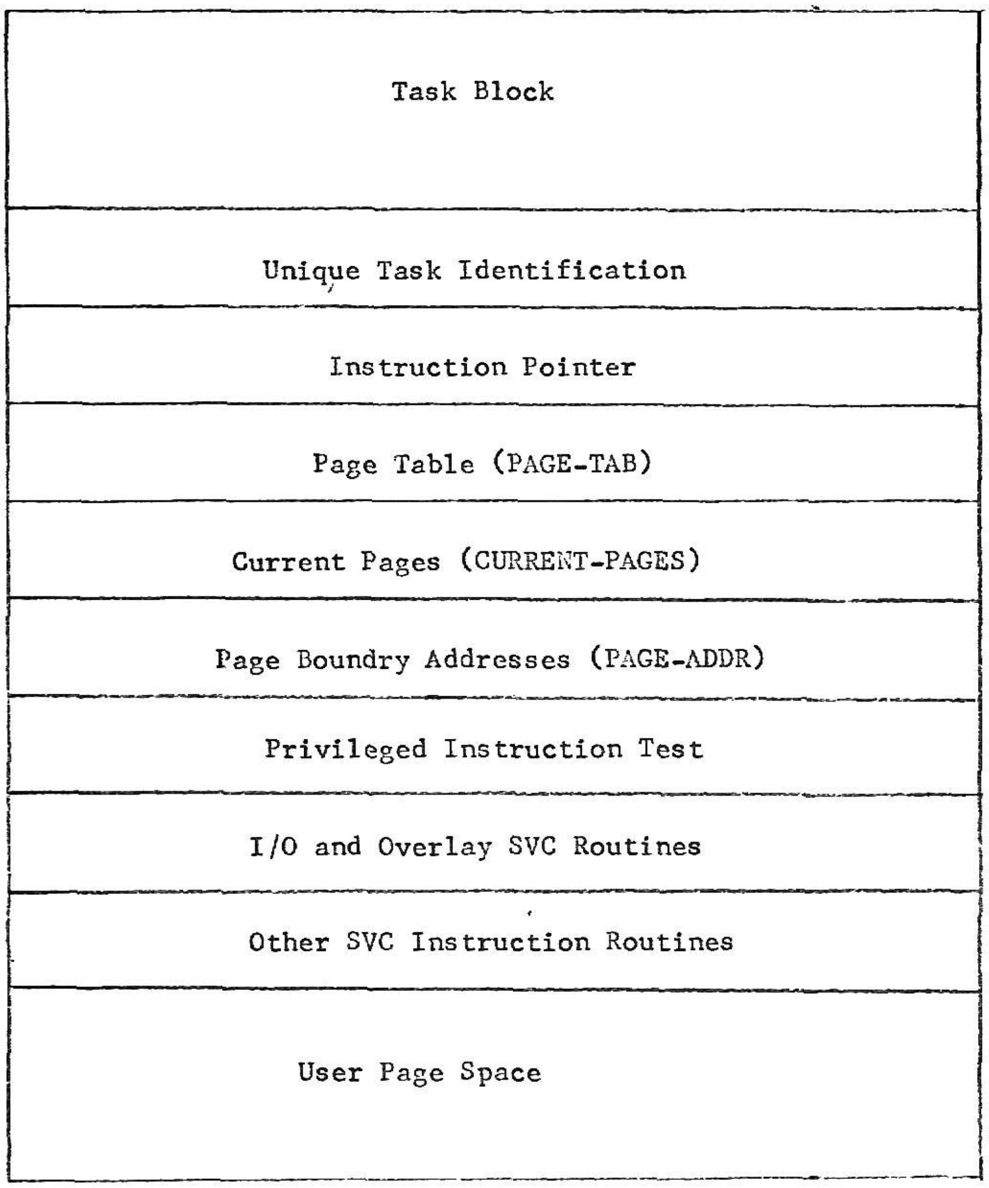

Task Block
Unique Task Identification
Instruction Pointer
Page Table (PAGE-TAB)
Current Pages (CURRENT-PAGES)
Page Boundry Addresses (PAGE-ADDR)
Privileged Instruction Test
I/O and Overlay SVC Routines
Other SVC Instruction Routines
User Page Space

Figure 3-4

user's parameter block will be concatenated with another parameter block shown in figure 3-3. The parameter block is then passed as a message to the Nova system by being used as data for an output SVC to the Nova device. The Nova system will evaluate the parameter block, and proceed with the I/O operation as directed by this parameter block. When an overlay SVC is received, the procedure is the same because the overlay operation is identical to a read and wait I/O instruction.

An end of job SVC means the present user must be deleted from the system and a new user brought in. The operating system must not be allowed to cancel the partition while this transfer is taking place. The parameter queue of the task will be prevented from becoming empty in order to keep the task from terminating. There will be messages sent to the Nova system specifying the procedure to transfer users in and out. These messages will be sent as data for an I/O SVC from the EXTOS to the Nova system. Thus through a basic capability of OS/16-MT, the problem of message communication becomes a simple I/O simulation.

A call task or cancel task SVC will cause a search for a unique task ID given by the instruction. The task ID of the PCBs in the task partitions are compared to the specified Task ID. For a call task, if the task is not located in the system a message will be sent to the Nova system, when a current task reaches end of job. This message will request that the next job to enter the system be the specified task. If the Nova is unable to fill the request, then the task issuing the call task SVC will be terminated. For a cancel task, if the specified task is not currently in the system, the instruction is ignored.

All other SVC instructions will be recreated and executed.

3.4 ISP (13) FOR INTERDATA 85 ASSEMBLER LANGUAGE

The incomplete ISP description for the host machine's assembler language is given in appendix B. Obvious deletions are the ISP for privileged, SVC, and floating point instructions. Privileged instructions will not be executed by the HIMICS system and are therefore not included. SVC instructions will be executed by the EXTOS so they are not included in the ISP descriptions either. Floating point instructions are not as yet included anywhere in the system. It is assummed they will be added to the emulator program and therefore will have an ISP description.

The ISP description is also incomplete in that there are many other considerations to be observed than just the basic results of the instruction. For example, the ISP for the ATL (add to the top of a list) instruction may be described with two or three ISP operations. The actual microcode routine which executes the instruction is nearly 20 operations long. This comparison is listed in appendix C. A lower level ISP is also given. This level may be easily translated to microcode. A comparison between the LM (load multiple) high-level ISP description, low level ISP description, and the actual microcode equivalent is also given in appendix C. The ISP for a LM instruction is comparable to the microcode routine in length. This is the case with many instructions.

3.5 HIMICS SYSTEM CONTROL PROCESS

The basic system is controlled by an emulator control program (appendix D). This control algorithm will process the instruction pointer in the EXTOS. Before executing an instruction, the page of the program which contains that instruction must be located in Level 1 Interdata memory. The virtual address of any memory reference operand must also be mapped to a real Level 1 memory address. Control is passed to the address mapper algorithm (ADDRMAP algorithm 2-7) to perform this conversion. While mapping the address, a page fault may occur. The algorithm which performs a NUR page removal scheme (FAULT algorithm 2-2) is given control. The page to be removed is transferred at the same time the requested page (the one which contains the virtual address which created the page fault) is brought into Level 1 Interdata memory. When the pages referenced by the instruction are in Level 1 memory, then the microcode routine which executes it is called, and the instruction is executed. The instruction pointer is adjusted as a process of the instruction execution. Therefore control returns to the emulator control process. This description applies to normal instructions (i.e. not SVC instructions). Figure 3-5 shows the contents of the emulation routine.

Emulator Contents

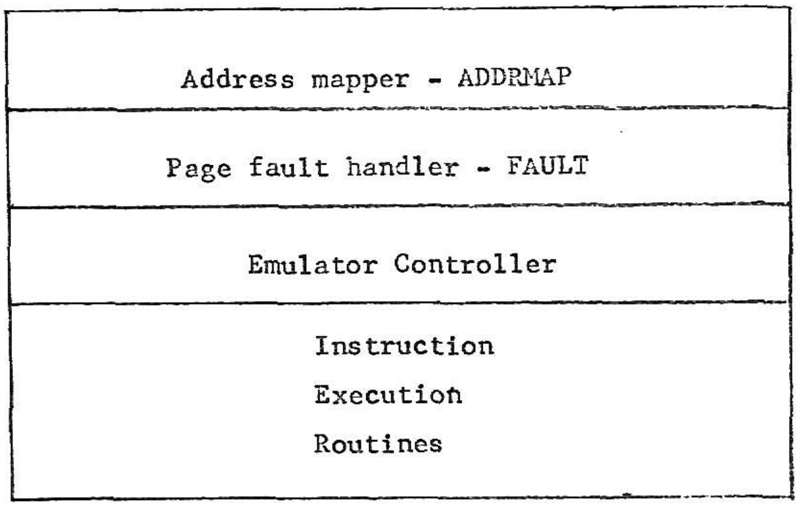


Figure 3-5

CHAPTER FOUR

4.1 INTRODUCTION

Chapter four will discuss future modification of HIMICS, and a concluding summary.

4.2 FUTURE WORK

There is a trememdous amount of detail which must be defined before HIMICS can be implemented. This section will not elaborate on this, but rather some major areas of modification will be discussed.

4.2.1 EMULATOR MODIFICATIONS

As stated earlier in this paper, the privileged instruction set is considered off limits for any task operating in the HIMICS system. In order to allow a user to legally execute a privileged instruction, the automatic fetch and decode of the Fixed Control Store will have to be overridden. This instruction will force control to reside in the FCS. If control is allowed to remain in the FCS, then the emulation process will be bypassed, and the program will terminate due to addressing errors.

An alternative method of allowing privileged instructions is to create the microcode which executes the instructions. This method would probably produce the best results in the long run once the new code was operational.

Floating point instructions are not included in the emulation description. There appear to be no problems which would keep them from being added to the system.

4.2.2 EXTENDED OPERATING SYSTEM MODIFICATIONS

One feature which will be added to the HIMICS system is the ability to change the size of the partitions in Level 1 memory. This change mostly affects the size and type of the data structures involved with keeping the page information needed for address mapping and page fault handling. The actual decisions concerning the size of the partitions will be made in the peripheral processor. This is to allow a maximum amount of user processing to be done in the Level 1 processor. The best approach is probably to use the same working set algorithms currently being documented (5) for the Level 2 processor. These algorithms would adjust the partition size in Level 1 memory at the same rate as Level 2 memory.

4.2.3 DEADLOCK

A possible deadlock situation exists in the HIMICS system due to the inter-task communication allowed by the host operating system. For example, if there were two tasks in the system, and both were in a task wait state, the system would be in a deadlock situation because a requested task cannot be brought into the Level 1 memory until another task goes to end of job. This kind of deadlock could be prevented by making sure there is at least one task which is not in the task wait state.

4.3 HIMICS SUMMARY

The HIMICS system is a hierarchical structure which provides a virtual memory environment for a network of mini-computers. The HIMICS system does not place any requirements on the hardware used to implement the system. The hierarchical design of HIMICS allows communication only between sequential levels in the system. This important design attribute allows the system to use a variable number of processors at the different levels of the hierarchy.

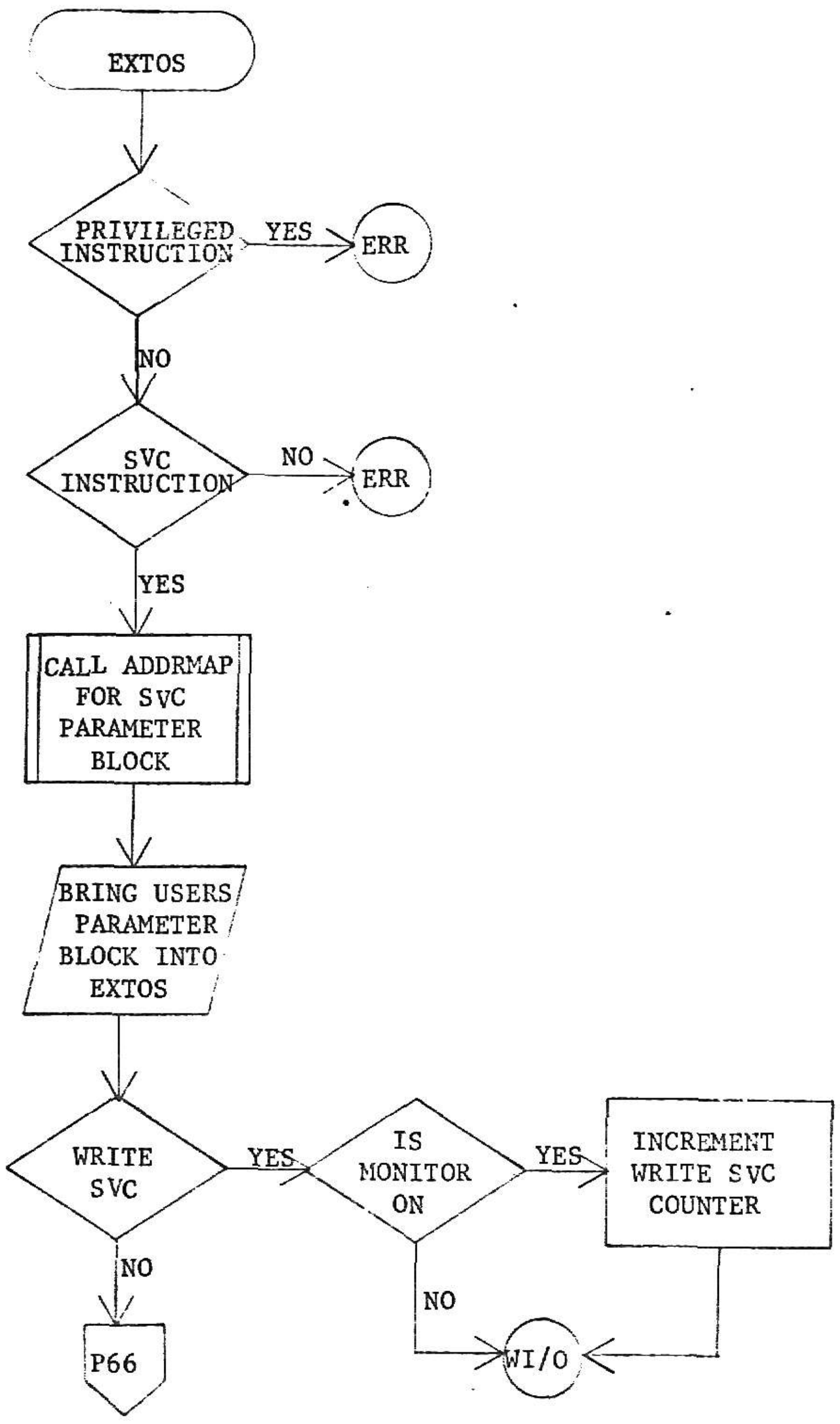
The implementation design should take advantage of HIMICS hierarchical structure. The Level 1 processor should be microprogramable in order to reduce the execution overhead of the emulation process. Depending upon the number of processors used, multi-tasking capabilities of the CPUs may be desired. This would result in faster execution and better throughput for the system.

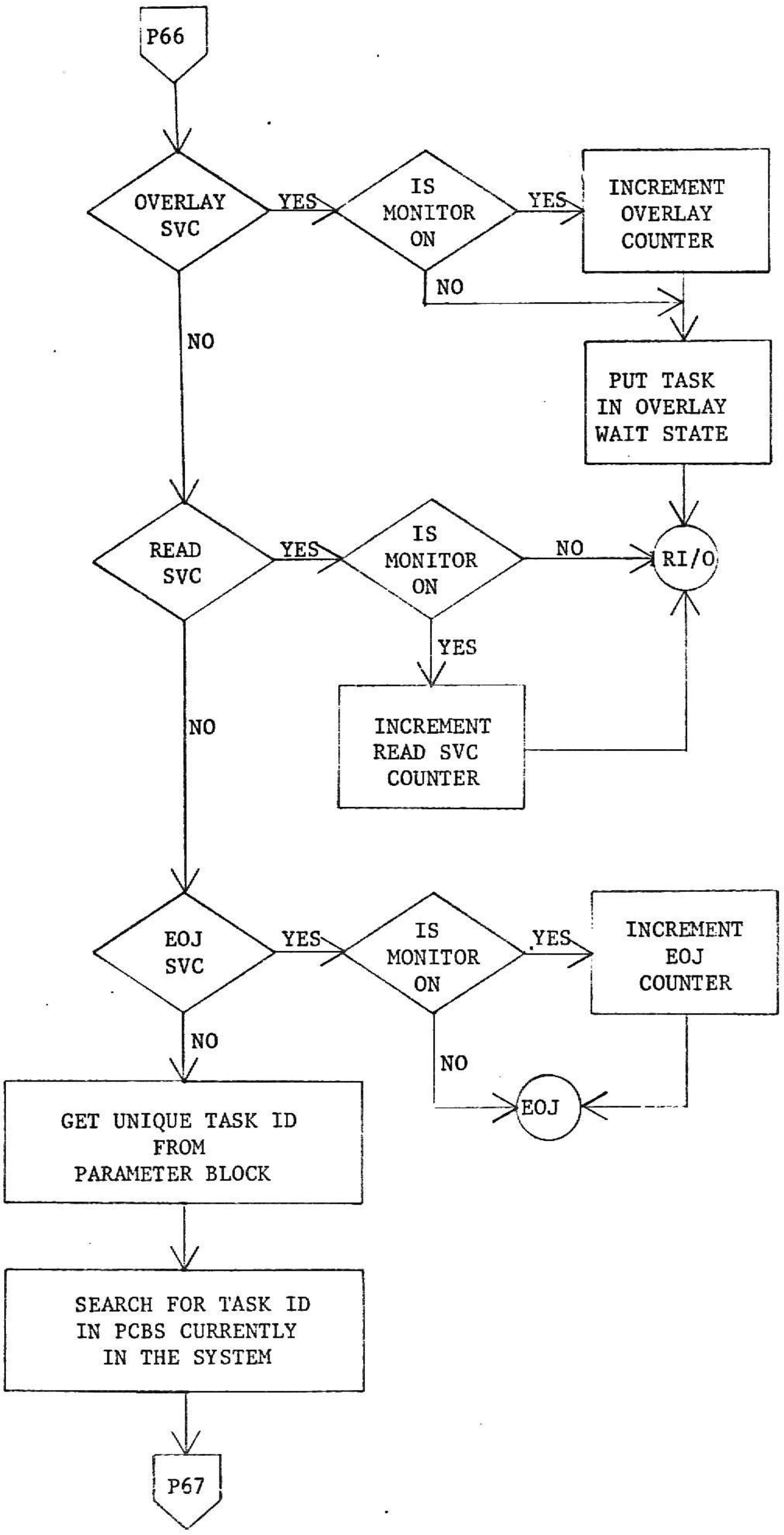
The implementation presented in this report is a design with two processors. The faster processor with multi-tasking capabilities is used as the Level 1 processor. The upper four levels of the HIMICS system will be located there. The system throughput is greatly affected by the processing speed of this processor and its microprogramming capabilities. As many functions as possible are placed in the secondary processor to also help increase system efficiency.

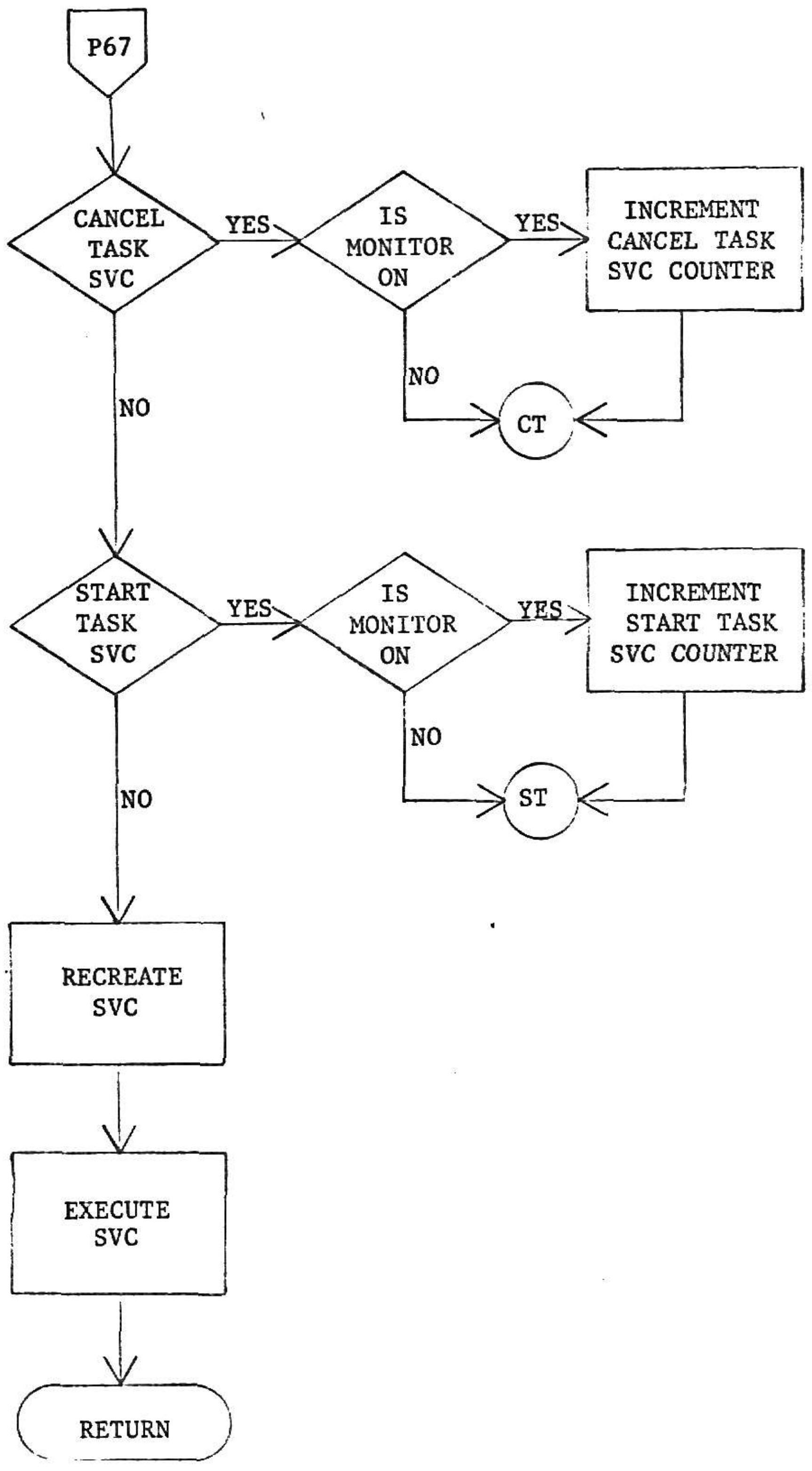
The HIMICS system is a virtual memory system. In many cases, this capability in itself would justify implementing the system. System throughput should also be increased. The user must decide whether or not the resulting benefits would justify the implementation of the system.

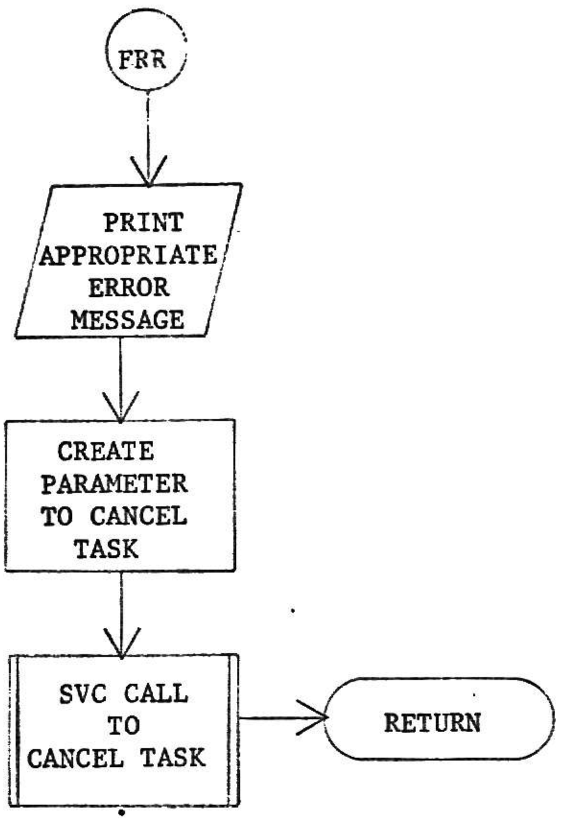
APPENDIX A

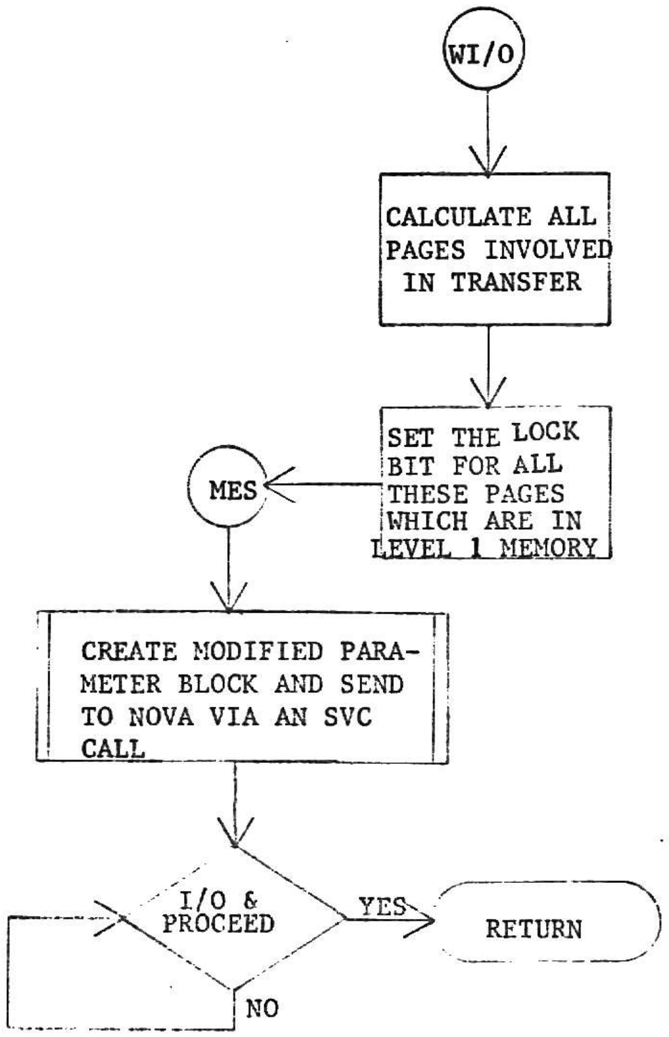
FLOWCHART OF THE EXTENDED OPERATING SYSTEM'S CONTROL

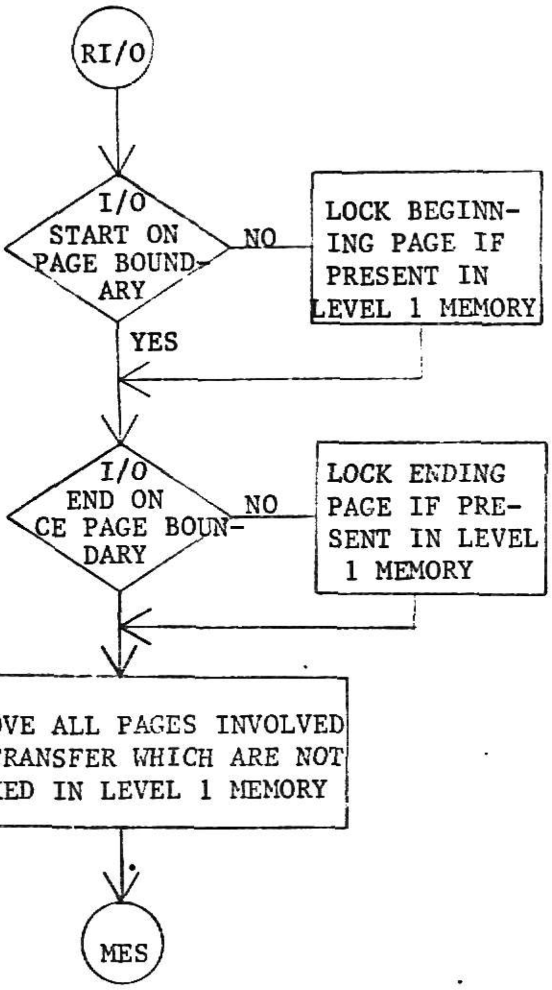

EXTOS
PRIVILEGED INSTRUCTION
YES
ERR
NO
SVC INSTRUCTION
NO
ERR
YES
CALL ADDRMAP FOR SVC PARAMETER BLOCK
BRING USERS PARAMETER BLOCK INTO EXTOS
WRITE SVC
YES
IS MONITOR ON
YES
INCREMENT WRITE SVC COUNTER
NO
P66
NO
WI/O

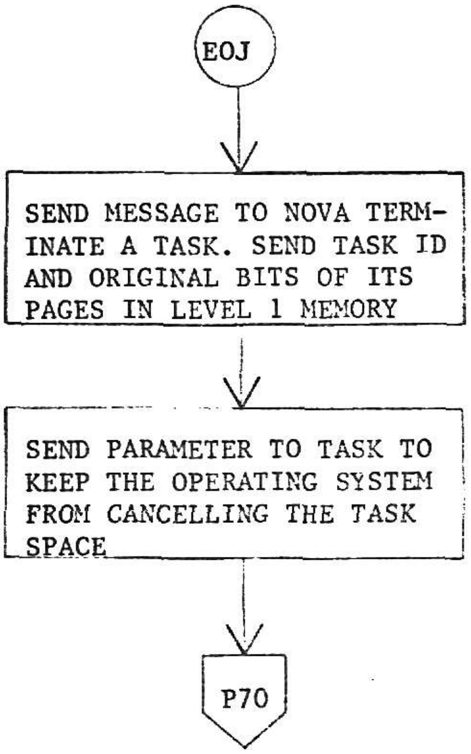

P66
OVERLAY SVC
YES
IS MONITOR ON
YES
INCREMENT OVERLAY COUNTER
NO
NO
PUT TASK IN OVERLAY WAIT STATE
READ SVC
YES
IS MONITOR ON
NO
RI/O
YES
NO
INCREMENT READ SVC COUNTER
EOJ SVC
YES
IS MONITOR ON
YES
INCREMENT EOJ COUNTER
NO
NO
EOJ
GET UNIQUE TASK ID FROM PARAMETER BLOCK
SEARCH FOR TASK ID IN PCBS CURRENTLY IN THE SYSTEM
P67

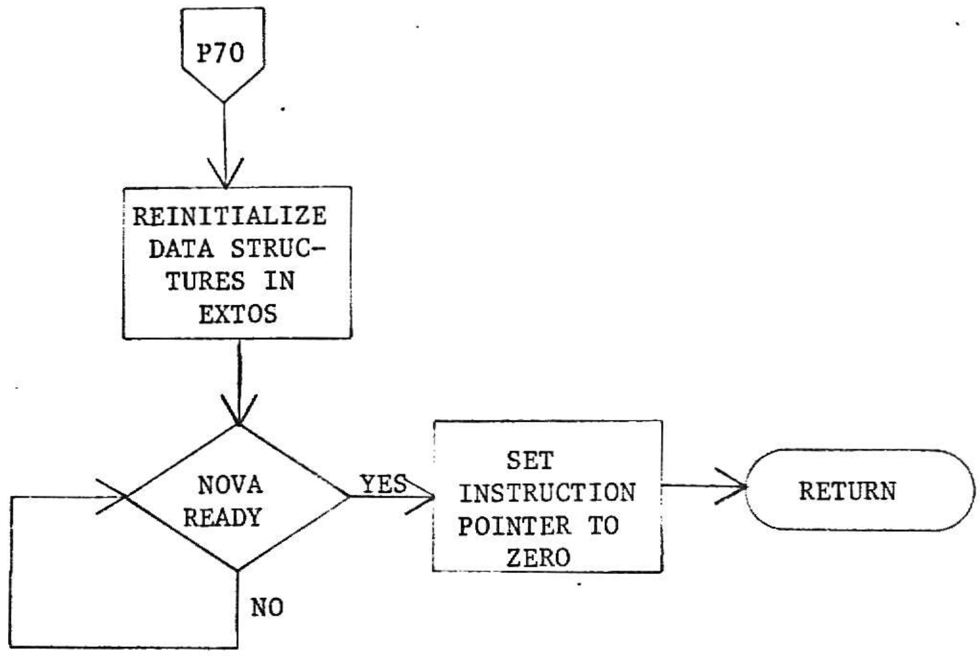

P67
CANCEL TASK SVC
YES
IS MONITOR ON
YES
INCREMENT CANCEL TASK SVC COUNTER
NO
CT
NO
START TASK SVC
YES
IS MONITOR ON
YES
INCREMENT START TASK SVC COUNTER
NO
ST
NO
RECREATE SVC
EXECUTE SVC
RETURN

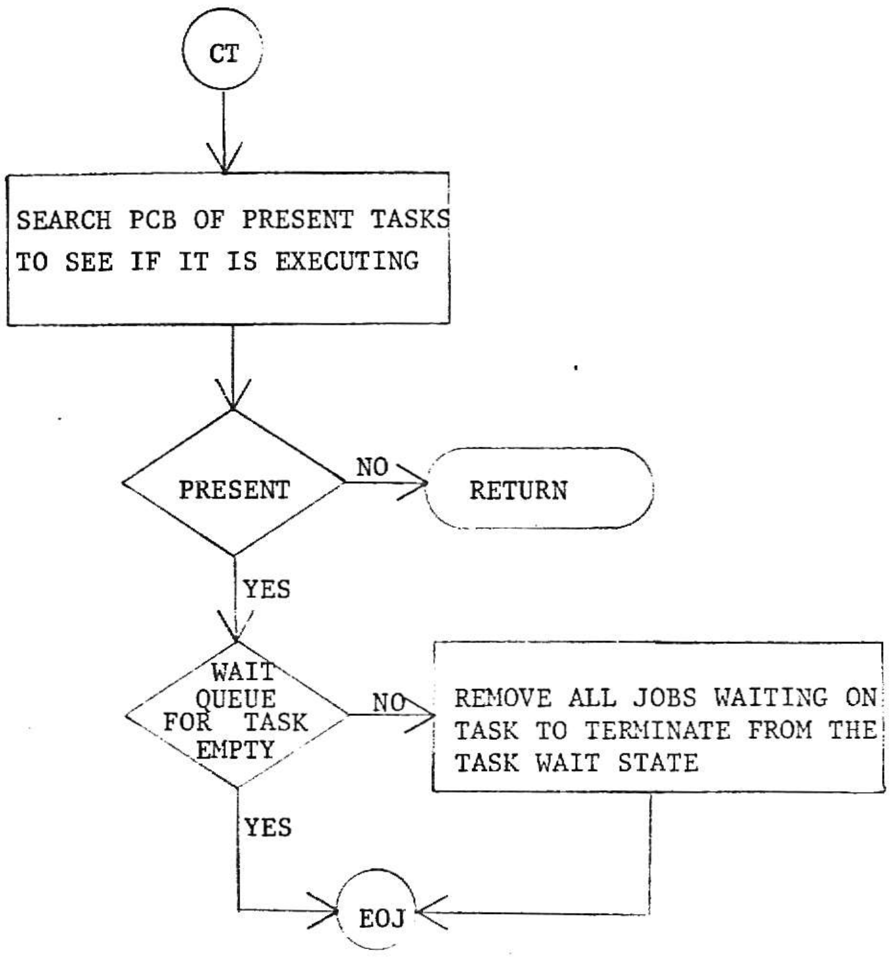

FRR
PRINT APPROPRIATE ERROR MESSAGE
CREATE PARAMETER TO CANCEL TASK
SVC CALL TO CANCEL TASK
RETURN

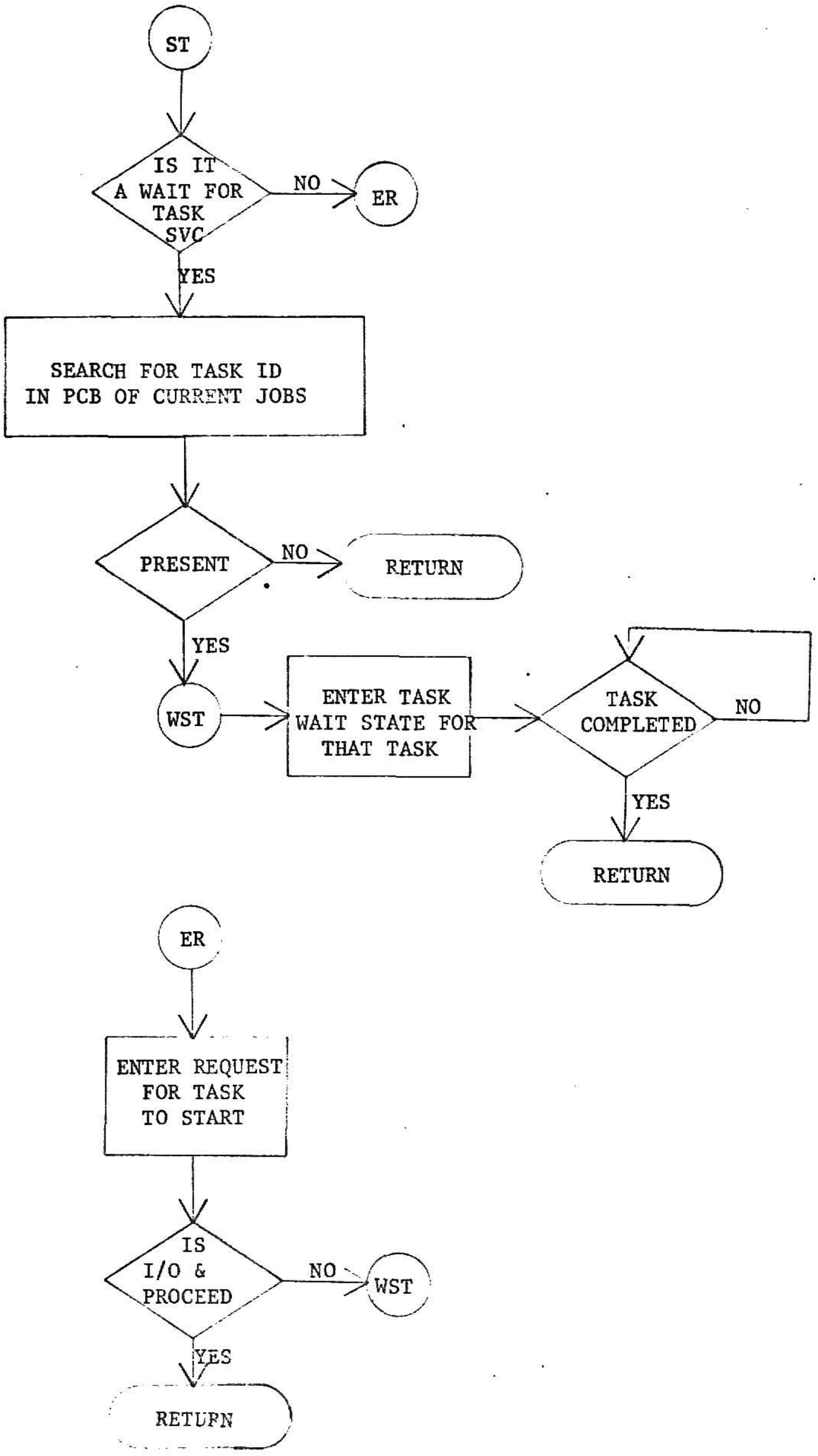

WI/O
CALCULATE ALL PAGES INVOLVED IN TRANSFER
SET THE LOCK BIT FOR ALL THESE PAGES WHICH ARE IN LEVEL 1 MEMORY
MES
CREATE MODIFIED PARAMETER BLOCK AND SEND TO NOVA VIA AN SVC CALL
I/O & PROCEED
YES
RETURN
NO


RI/O
I/O START ON PAGE BOUNDARY
NO
YES
LOCK BEGINNING PAGE IF PRESENT IN LEVEL 1 MEMORY
I/O END ON CE PAGE BOUNDARY
NO
LOCK ENDING PAGE IF PRESENT IN LEVEL 1 MEMORY
REMOVE ALL PAGES INVOLVED IN TRANSFER WHICH ARE NOT LOCKED IN LEVEL 1 MEMORY
MES


EOJ
SEND MESSAGE TO NOVA TERMINATE A TASK. SEND TASK ID AND ORIGINAL BITS OF ITS PAGES IN LEVEL 1 MEMORY
SEND PARAMETER TO TASK TO KEEP THE OPERATING SYSTEM FROM CANCELLING THE TASK SPACE
P70


P70
REINITIALIZE DATA STRUCTURES IN EXTOS
NOVA READY
YES
NO
SET INSTRUCTION POINTER TO ZERO
RETURN


CT
SEARCH PCB OF PRESENT TASKS TO SEE IF IT IS EXECUTING
PRESENT
NO
RETURN
YES
WAIT QUEUE FOR TASK EMPTY
NO
REMOVE ALL JOBS WAITING ON TASK TO TERMINATE FROM THE TASK WAIT STATE
YES
EOJ


ST
IS IT A WAIT FOR TASK SVC
NO
ER
YES
SEARCH FOR TASK ID IN PCB OF CURRENT JOBS
PRESENT
NO
RETURN
YES
WST
ENTER TASK WAIT STATE FOR THAT TASK
TASK COMPLETED
NO
YES
RETURN
ER
ENTER REQUEST FOR TASK TO START
IS I/O & PROCEED
NO
WST
YES
RETURN

APPENDIX B

ISP DESCRIPTION OF INTERDATA 85 MACHINE LANGUAGE EMULATOR

```
INSTRUCTION FORMAT:     INSTRUCTION [0:1] / I<0:15>

OPCODE   := I[0]<0:7>
R1,M1    := I[0]<8:11>
N,R2,X2,D := I[0]<12:15>
A        := I[1]<0:15>


REGISTERS:

PROGRAM STATUS WORD [0:1] / PSW<0:15>
STATUS  := PSW[0]<0:11>
CC      := PSW[0]<12:15>
C       := PSW[0]<12>
V       := PSW[0]<13>
G       := PSW [0]<14>
L       := PSW[0]<15>
LOC     := PSW[1]<0:15>

GENERAL REGISTERS [0:15] / GR<0:15>


SYMBOL DEFINITION:

AND := ∧ → 0∧0=0
           0∧1=0
           1∧0=0
           1∧1=1

OR := ∨ → 0∨0=0
          0∨1=1
          1∨0=1
          1∨1=1

EXCLUSIVE OR := ∇ → 0∇0=0
                    0∇1=1
                    1∇0=1
                    1∇1=0


INTERMEDIATE STORAGE:

REAL ADDRESS := REALADDR<0:15>
REMAINDER := REMAINDER<0:15>
NUMBER OF SLOTS IN THE LIST := NUMSLOT<0:7>
NUMBER OF SLOTS USED := USEDSLOTS<8:15>
CURRENT TOP := CURTOP<0:7>
NEXT BOTTOM := NEXTBOT<8:15>
CONDITION CODE ANDED WITH M1 := CCNM1<0:3>
THE BITS OF CCNM1 ORED TOGETHER := ORRESULT<0>
```

```
/*  FIXED-POINT LOAD AND STORE INSTRUCTIONS                                    */
/*  LIS: LOAD IMMEDIATE SHORT                                                  */
/*  LCS: LOAD COMPLEMENT SHORT                                                 */
/*  LHR: LOAD HALFWORD RR                                                      */
/*  LH:  LOAD HALFWORD                                                         */
/*  LHI: LOAD HALFWORD IMMEDIATE                                               */

LIS:   OPCODE  := 24 → (GR[R1]←N;RETURN;);
LCS:   OPCODE  := 25 → (GR[R1]←-N;RETURN;);
LHR:   OPCODE  := 08 → (GR[R1]←GR[R2];RETURN;);
LH:    OPCODE  := 48 → (REALADDR←ADDRMAP(A+GR[X2]);
                         GR[R1]←[REALADDR];RETURN;);
LHI:   OPCODE  := C8 → (GR[R1]←A+GR[X2];RETURN;);

/*  LM:  LOAD MULTIPLE                                                         */
/*  STH: STORE HALFWORD                                                        */
/*  STM: STORE MULTIPLE                                                        */

LM:    OPCODE  := D1 → (REALADDR←ADDRMAP(A+GR[X2]);
                         GR[R1]←[REALADDR];
                         R1=15→RETURN;
                         R1≠15→(R1←R1+1;A←A+2);→LM;);
STH:   OPCODE  := 40 → (REALADDR←ADDRMAP(A+GR[X2]);
                         GR[R1]→REALADDR;RETURN;);
STM:   OPCODE  := D0 → (REALADDR←ADDRMAP(A+GR[X2]);
                         GR[R1]→REALADDR;
                         R1=15→RETURN;
                         R1≠15→(R1←R1+1;A←A+2);→STM;);

/*  FIXED-POINT ARITHMETIC INSTRUCTIONS                                        */
/*  ADD AND SUBSTRUCT                                                          */
/*  AIS: ADD IMMEDIATE SHORT                                                   */
/*  AHR: ADD HALFWORD RR                                                       */
/*  AH:  ADD HALFWORD                                                          */
/*  AHI: ADD HALFWORD IMMEDIATE                                                */
/*  AHM: ADD HALFWORD TO MEMORY                                                */
/*  ACHR: ADD WITH CARRY HALFWORD RR                                           */
/*  ACH: ADD WITH CARRY HALFWORD                                               */

AIS:   OPCODE  := 26 → (GR[R1]←GR[R1]+N;RETURN;);
AHR:   OPCODE  := 0A → (GR[R1]←GR[R1]+GR[R2];RETURN;);
AH:    OPCODE  := 4A → (REALADDR←ADDRMAP(A+GR[X2]);
                         GR[R1]←GR[R1]+[REALADDR];RETURN;);
AHI:   OPCODE  := CA → (GR[R1]←GR[R1]+A+GR[X2];RETURN;);
AHM:   OPCODE  := 61 → (REALADDR←ADDRMAP(A+GR[X2]);
                         [REALADDR]←GR[R1]+[REALADDR];RETURN;);
ACHR:  OPCODE  := 0E → (GR[R1]←GR[R1]+GR[R2]+C;RETURN;);
ACH:   OPCODE  := 4E → (REALADDR←ADDRMAP(A+GR[X2]);
                         GR[R1]←GR[R1]+[REALADDR]+C;RETURN;);
```

```
/*  SIS: SUBTRACT IMMEDIATE SHORT                                     */
/*  SHR  SUBTRACT HALFWORD RR                                         */
/*  SH:  SUBTRACT HALFWORD                                            */
/*  SHI: SUBTRACT HALFWORD IMMEDIATE                                  */
/*  SCHR: SUBTRACT WITH CARRY HALFWORD RR                             */
/*  SCH: SUBTRACT WITH CARRY HALFWORD                                 */

SIS:  OPCODE := 27 → (GR[R1]←GR[R1]-N;RETURN;);
SHR:  OPCODE := 0B → (GR[R1]←GR[R1]-GR[R2];RETURN;);
SH:   OPCODE := 4B → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1]←GR[R1]-[REALADDR];RETURN;);
SHI:  OPCODE := CB → (GR[R1]←GR[R1]-A-GR[X2];RETURN;);
SCHR: OPCODE := 0F → (GR[R1]←GR[R1]-GR[R2]-C;RETURN;);
SCH:  OPCODE := 4F → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1]←GR[R1]-[REALADDR]-C;RETURN;);

/*  COMPARE INSTRUCTIONS                                              */
/*  CLHR: COMPARE LOGICAL HALFWORD RR                                 */
/*  CLH: COMPARE LOGICAL HALFWORD                                     */
/*  CLHI: COMPARE LOGICAL HALFWORD IMMEDIATE                          */
/*  CHR: COMPARE HALFWORD RR                                          */
/*  CH:  COMPARE HALFWORD                                             */
/*  CHI: COMPARE HALFWORD IMMEDIATE                                   */

CLHR: OPCODE := 05 → (GR[R1]:GR[R2];RETURN;);
CLH:  OPCODE := 45 → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1]:[REALADDR];RETURN;);
CLHI: OPCODE := C5 → (GR[R1]:A+GR[X2];RETURN;);
CHR:  OPCODE := 09 → (GR[R1]:GR[R2];RETURN;);
CH:   OPCODE := 49 → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1]:[REALADDR];RETURN;);
CHI:  OPCODE := C9 → (GR[R1]:A+GR[X2];RETURN;);

/*  MULTIPLY AND DIVIDE INSTRUCTIONS                                  */
/*  MHR: MULTIPLY HALFWORD RR                                         */
/*  MH:  MULTIPLY HALFWORD                                            */
/*  MHUR: MULTIPLY HALFWORD UNSIGNED RR                               */
/*  MHU: MULTIPLY HALFWORD UNSIGNED                                   */
/*  DHR: DIVIDE HALFWORD RR                                           */
/*  DH:  DIVIDE HALFWORD                                              */

MHR:  OPCODE := 0C → (GR[R1,R1+1]←GR[R1+1]*GR[R2];RETURN;);
MH:   OPCODE := 4C → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1,R1+1]←GR[R1+1]*[REALADDR];RETURN;);
MHUR: OPCODE := 9C → (GR[R1,R1+1]←GR[R1+1]*GR[R2];RETURN;);
MHU:  OPCODE := DC → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1,R1+1]←GR[R1+1]*[REALADDR];RETURN;);
DHR:  OPCODE := 0D → (GR[R1+1]←GR[R1,R1+1]/GR[R2];
                      GR[R1]←REMAINDER;RETURN;);
DH:   OPCODE := 4D → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1+1]←GR[R1,R1+1]/[REALADDR];
                      GR[R1]←REMAINDER;RETURN;);
```

```
/*   LOGICAL AND BIT MANIPULATING INSTRUCTIONS                                    */
/*   NHR: AND HALFWORD RR                                                         */
/*   NH:  AND HALFWORD                                                            */
/*   NHI: AND HALFWORD IMMEDIATE                                                  */
/*   OHR: OR HALFWORD RR                                                          */
/*   OH:  OR HALFWORD                                                             */
/*   OHI: OR HALFWORD IMMEDIATE                                                   */
/*   XHR:  EXCLUSIVE OR HALFWORD RR                                               */
/*   XH:  EXCLUSIVE OR HALFWORD                                                   */
/*   XHI: EXCLUSIVE OR HALFWORD IMMEDIATE                                         */

NHR:  OPCODE := 04 → (GR[R1]←GR[R1]∧GR[R2];RETURN;);
NH:   OPCODE := 44 → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1]←GR[R1]∧[REALADDR];RETURN;);
NHI:  OPCODE := C4 → (GR[R1]←GR[R1]∧A+GR[X2];RETURN;);
OHR:  OPCODE := 06 → (GR(R1]←GR[R1]∨GR[R2];RETURN;);
OH:   OPCODE := 46 → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1]←GR[R1]∨[REALADDR];RETURN;);
OHI:  OPCODE := C6 → (GR[R1]←GR[R1]∨A+GR[X2];RETURN;);
XHR:  OPCODE := 07 → (GR[R1]←GR[R1]∇GR[R2];RETURN;);
XH:   OPCODE := 47 → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1]←GR[R1]∇[REALADDR];RETURN;);
XHI:  OPCODE := C7 → (GR[R1]←GR[R1]∇A+GR[X2];RETURN;);

/*   THI: TEST HALFWORD IMMEDIATE                                                 */

THI:  OPCODE := C3 → (GR[R1]∧A+GR[X2];RETURN;);

/*   BYTE HANDLING INSTRUCTIONS                                                   */
/*   LBR: LOAD BYTE RR                                                            */
/*   LB:  LOAD BYTE                                                               */
/*   STBR: STORE BYTE RR                                                          */
/*   STB: STORE BYTE                                                              */
/*   EXBR: EXCHANGE BYTE                                                          */
/*   CLB: COMPARE LOGICAL BYTE                                                    */

LBR:  OPCODE := 93 → (GR[R1<8:15>]←GR[R2<8:15>];
                      GR[R1<0:7>]←0;RETURN;);
LB:   OPCODE := D3 → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1<8:15>]←[REALADDR];
                      GR[R1<0:7>]←0;RETURN;);
STBR: OPCODE := 92 → (GR[R1<8:15>]→GR[R1<8:15>];RETURN;);
STB:  OPCODE := D2 → (REALADDR←ADDRMAP(A+GR[X2]);
                      GP[R1<8:15>]→[REALADDR];RETURN;);
EXBR: OPCODE := 94 → (GR[R1<0:7>]←GR[R2<8:15>];
                      GR[R1<8:15>]←GR[R2<0:7>];RETURN;);
CLB:  OPCODE := D4 → (REALADDR←ADDRMAP(A+GR[X2]);
                      GR[R1<8:15>]:[REALADDR];RETURN;);
```

```
/*  LIST PROCESSING INSTRUCTIONS                                   */
/*  ATL: ADD TO TOP OF LIST                                        */
/*  ABL: ADD TO BOTTOM OF LIST                                     */
/*  RTL: REMOVE FROM TOP OF LIST                                   */
/*  RBL: REMOVE FROM BOTTOM OF LIST                                */

ATL:   OPCODE  := 64 → (NUMSLOTS-USEDSLOTS>0→(USEDSLOTS←USEDSLOTS+1;
                        CURTOP←CURTOP-1;[CURTOP]←GR[R1];);RETURN;);
ABL:   OPCODE  := 65 → (NUMSLOTS-USEDSLOTS>0→(USEDSLOTS←USEDSLOTS+1;
                        NEXTBOT←NEXTBOT+1;[NEXTBOT]←GR[R1];);RETURN;);
RTL:   OPCODE  := 66 → (USEDSLOTS>0→(USEDSLOTS←USEDSLOTS-1;
                        GR[R1]←[CURTOP];CURTOP←CURTOP+1;);RETURN;);
RBL:   OPCODE  := 67 → (USEDSLOTS>0→(USEDSLOTS←USEDSLOTS-1;
                        GR[R1]←[NEXTBOT];NEXTBOT←NEXTBOT-1;);RETURN;);

/*  SHIFT AND ROTATE INSTRUCTIONS                                  */
/*  SLLS: SHIFT LEFT LOGICAL SHORT                                 */
/*  SLHL: SHIFT LEFT HALFWORD LOGICAL                              */
/*  SLL:  SHIFT LEFT LOGICAL                                       */
/*  SRLS: SHIFT RIGHT LOGICAL SHORT                                */
/*  SRHL: SHIFT RIGHT HALFWORD LOGICAL                             */
/*  SRL:  SHIFT RIGHT LOGICAL                                      */

SLLS:  OPCODE  := 91 → (GR[R1]←GR[R1]*2**N;RETURN;);
SLHL:  OPCODE  := CD → (GR[R1]←GR[R1]*2**(A+GR[X2]<12:15>);RETURN;);
SLL:   OPCODE  := ED → (GR[R1,R1+1]←GR[R1,R1+1]*2**(A+GR[X2]<12:15>);
                        RETURN;);
SRLS:  OPCODE  := 90 → (GR[R1]←GR[R1]/2**N;RETURN;);
SRHL:  OPCODE  := CC → (GR[R1]←GR[R1]/2**(A+GR[X2]<12:15>);RETURN;);
SRL:   OPCODE  := EC → (GR[R1,R1+1]←GR[R1,R1+1]/2**(A+GR[X2]<12:15>);
                        RETURN;);

/*  RLL: ROTATE LEFT LOGICAL                                       */
/*  RRL: ROTATE RIGHT LOGICAL                                      */

RLL:   OPCODE  := EB → (GR[R1,R1+1]←GR[R1,R1+1]
                        *2**(A+GR[X2]<11:15>)[ROTATE];RETURN;);
RRL:   OPCODE  := EA → (GR[R1,R1+1]←GR[R1,R1+1]
                        /2**(A+GR[X2]<11:15>)[ROTATE];RETURN;);

/*  SLHA: SHIFT LEFT HALFWORD ARITHMETIC                           */
/*  SLA:  SHIFT LEFT ARITHMETIC                                    */
/*  SRHA: SHIFT RIGHT HALFWORD ARITHMETIC                          */
/*  SRA:  SHIFT RIGHT ARITHMETIC                                   */

SLHA:  OPCODE  := CF → (GR[R1]←GR[R1]*2**(A+GR[X2]<12:15>);RETURN;);
SLA:   OPCODE  := EF → (GR[R1,R1+1]←GR[R1,R1+1]
                        *2**(A+GR[X2]<11:15>);RETURN;);
SRHA:  OPCODE  := CE → (GR[R1]←GR[R1]/2**(A+GR[X2]<12:15>);RETURN;);
SRA:   OPCODE  := EE → (GR[R1,R1+1]←GR[R1,R1+1]
                        /2**(A+GR[X2]<11:15>);RETURN;);
```

```
/*  BRANCH INSTRUCTIONS                                                */
/*  BTBS: BRANCH TRUE BACKWARD SHORT                                   */
/*  BTFS: BRANCH TRUE FORWARD SHORT                                    */
/*  BTCR: BRANCH TRUE CONDITION RR                                     */
/*  BTC:  BRANCH TRUE CONDITION                                        */
/*  BFBS: BRANCH FALSE BACKWARD SHORT                                  */
/*  BFFS: BRANCH FALSE FORWARD SHORT                                   */
/*  BFCR: BRANCH FALSE CONDITION RR                                    */
/*  BFC:  BRANCH FALSE CONDITION                                       */
/*  BXH:  BRANCH INDEX HIGH                                            */
/*  BXLE: BRANCH INDEX LOW OR EQUAL                                    */
/*  BALR: BRANCH AND LINK RR                                           */
/*  BAL:  BRANCH AND LINK                                              */

BTBS: OPCODE := 20 → (ORRESULT=1→(LOC←LOC-(2*D);RETURN;)
                      ORRESULT=0→(LOC←LOC+2;RETURN;));
BTFS: OPCODE := 21 → (ORRESULT=1→(LOC←LOC+(2*D);RETURN;)
                      ORRESULT=0→(LOC←LOC+2;RETURN;));
BTCR: OPCODE := 02 → (ORRESULT=1→(LOC←GR[R2];RETURN;)
                      ORRESULT=0→(LOC←LOC+2;RETURN;));
BTC:  OPCODE := 42 → (ORRESULT=1→(LOC←A+GR[X2];RETURN;)
                      ORRESULT=0→(LOC←LOC+4;RETURN;));
BFBS: OPCODE := 22 → (ORRESULT=0→(LOC←LOC-(2*D);RETURN;)
                      ORRESULT=1→(LOC←LOC+2;RETURN;));
BFFS: OPCODE := 23 → (ORRESULT=0→(LOC←LOC+(2*D);RETURN;)
                      ORRESULT=1→(LOC←LOC+2;RETURN;));
BFCR: OPCODE := 03 → (ORRESULT=0→(LOC←GR[R2];RETURN;)
                      ORRESULT=1→(LOC←LOC+2;RETURN;));
BFC:  OPCODE := 43 → (ORRESULT=0→(LOC←A+GR[X2];RETURN;)
                      ORRESULT=1→(LOC←LOC+4;RETURN;));
BXH:  OPCODE := C0 → (GR[R1]←GR[R1]+GR[R1+1];
                      (GR[R1]-GR[R1+1])>0→(LOC←A+GR[X2];RETURN;)
                      (GR[R1]-GR[R1+1])≤0→(LOC←LOC+4;RETURN;));
BXLE: OPCODE := C1 → (GR[R1]←GR[R1]+GR[R1+1];
                      (GR[R1]-GR[R1+1])≤0→(LOC←A+GR[X2];RETURN;)
                      (GR[R1]-GR[R1+1])>0→(LOC←LOC+4;RETURN;));
BALR: OPCODE := 01 → (GR[R1]←LOC+2;LOC←GR[R2];RETURN;);
BAL:  OPCODE := 41 → (GR[R1]←LOC+4;LOC←A+GR[X2];RETURN;);
```

APPENDIX C

CONTAINED IN THIS APPENDIX IS A COMPARISON BETWEEN TWO LEVELS OF ISP DESCRIPTION AND THE MICROCODE FOR A LOAD MULTIPLE AND AN ADD TO TOP OF LIST INSTRUCTIONS.

HIGH LEVEL ISP FOR A LOAD MULTIPLE INSTRUCTION:

```
LM:   OPCODE  := D1  → (REALADDR←ADDRMAP(A+GR[X2]);
                         GR[R1]←[REALADDR];
                         R1=15→RETURN;
                         R1≠15→(R1←R1+1;A←A+2);→LM;);
```

LOW LEVEL ISP FOR A LOAD MULTIPLE INSTRUCTION USING THE MICROCODE VARIABLES AND REGISTERS:

```
LM:   OPCODE  := D1  → (MAR←[MDR]+[YX];
                         MR1←MAR;
                         MR0←-15;
LM1:                     MAR←ADDRMAP([MR1]);
                         YD←[MDR];
                         MR1←MR1+2;
                         YDI←YDI+MR0;
                         YDI≠0→(YD←YD+1;→LM1;);
                         YDI=0→(RETURN););
```

MICROCODE FOR A LOAD MULTIPLE INSTRUCTION:

```
LM:     A     MAR,MDR,YX,DR4        CALCULATE STARTING VIRTUAL ADDR
        L     MR1,MAR               KEEP THE INCREASING VIRTUAL
                                    ADDRESS IN MR1
        LI    MR0,'FFF1'            PUT -15 IN MR0
LM1:    BAL   ADDRMAP(MR7)          THE REAL ADDRESS OF THE VIRTUAL
                                    ADDRESS CONTAINED IN MR1 WILL
                                    BE PUT IN MAR
        L     YD,MDR,DR2            MDR HAS CONTENTS OF LOCATION MAR
        AI    MR1,MR1,'0002'        VIRTUAL ADDRESS BUMPED BY TWO
        AX    YDI,YDI,MR0,LM1,C     TEST FOR REGISTER 15
        BAL   (MR6)(MR7)            MR6 HAS RETURN ADDRESS TO
                                    EMULATOR CONTROL PROGRAM
```

HIGH LEVEL ISP FOR AN ADD TO TOP OF LIST INSTRUCTION(DOES NOT INCLUDE OVERFLOW ERROR CONDITION):

```
ATL: OPCODE  := 64 →(NUMSLOTS-USEDSLOTS>0→(USEDSLOTS←USEDSLOTS+1;
                     CURTOP←CURTOP-1;[CURTOP]←GR[R1];);RETURN;
                     NUMSLOTS-USEDSLOTS≤0→OVERFLOW;);
```

LOW LEVEL ISP FOR AN ADD TO TOP OF LIST INSTRUCTION(DOES NOT INCLUDE OVERFLOW ERROR CONDITION):

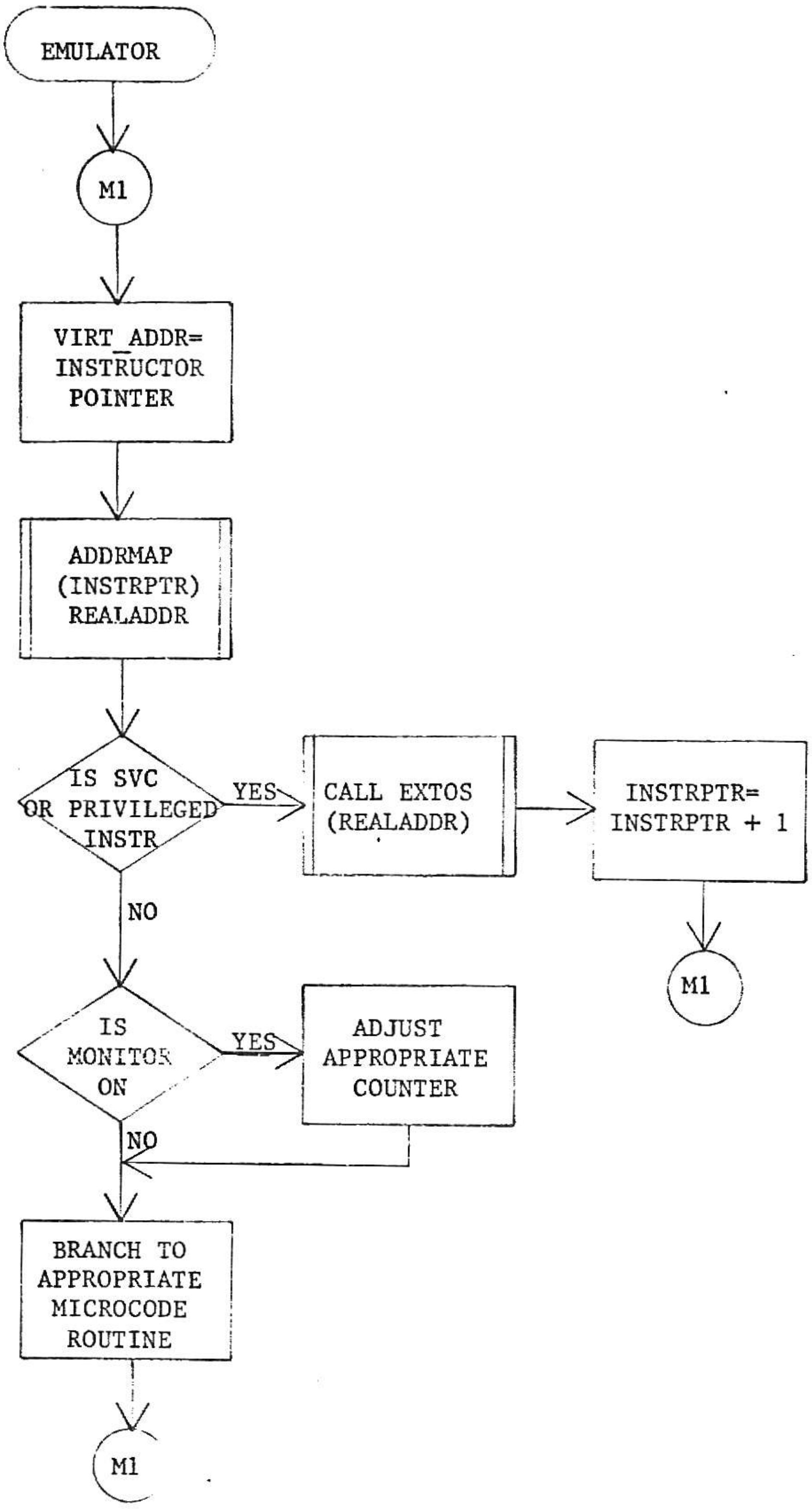
```
ATL: OPCODE := 64 → (MR1←[MDR]+[YX];
                     MAR←ADDRMAP([MR1]);
                     MR5←1;MR2←MDR<0:7>;
                     MR3←MDR<8:15>;
                     MR3≥MR2→OVERFLOW;
                     MR3<MR2→(MDR←MDR+MR5;
                     MR1←MR1+2;MAR←ADDRMAP([MR1]);
                     MR4←MDR<0:7>;MR4=0→(MR4←MR2);
                     MR4←MR4+MR4;MR1←MR1+MR4;
                     MAR←ADDRMAP([MR1]);MDR←[YD];RETURN;););
```

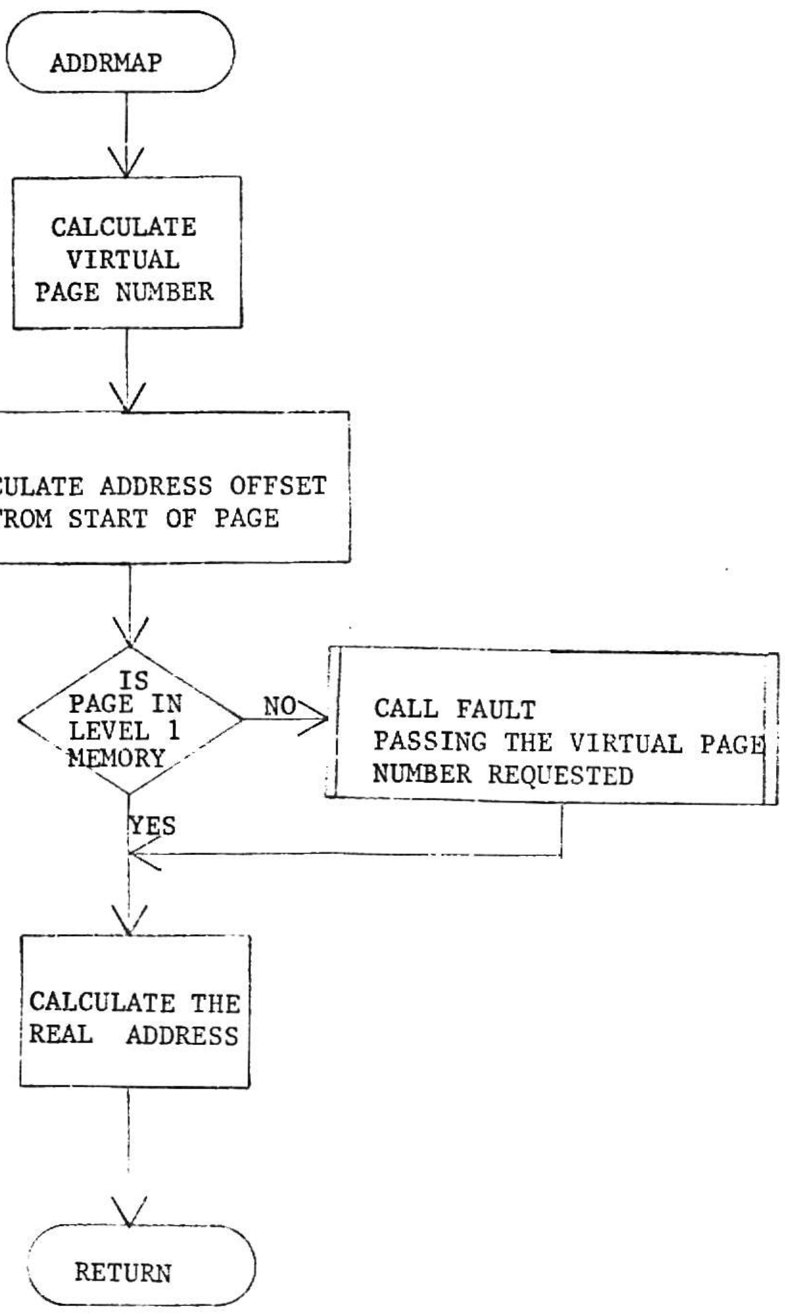
MICROCODE FOR AN ADD TO TOP OF LIST INSTRUCTION(DOES NOT INCLUDE OVERFLOW ERROR CONDITION):

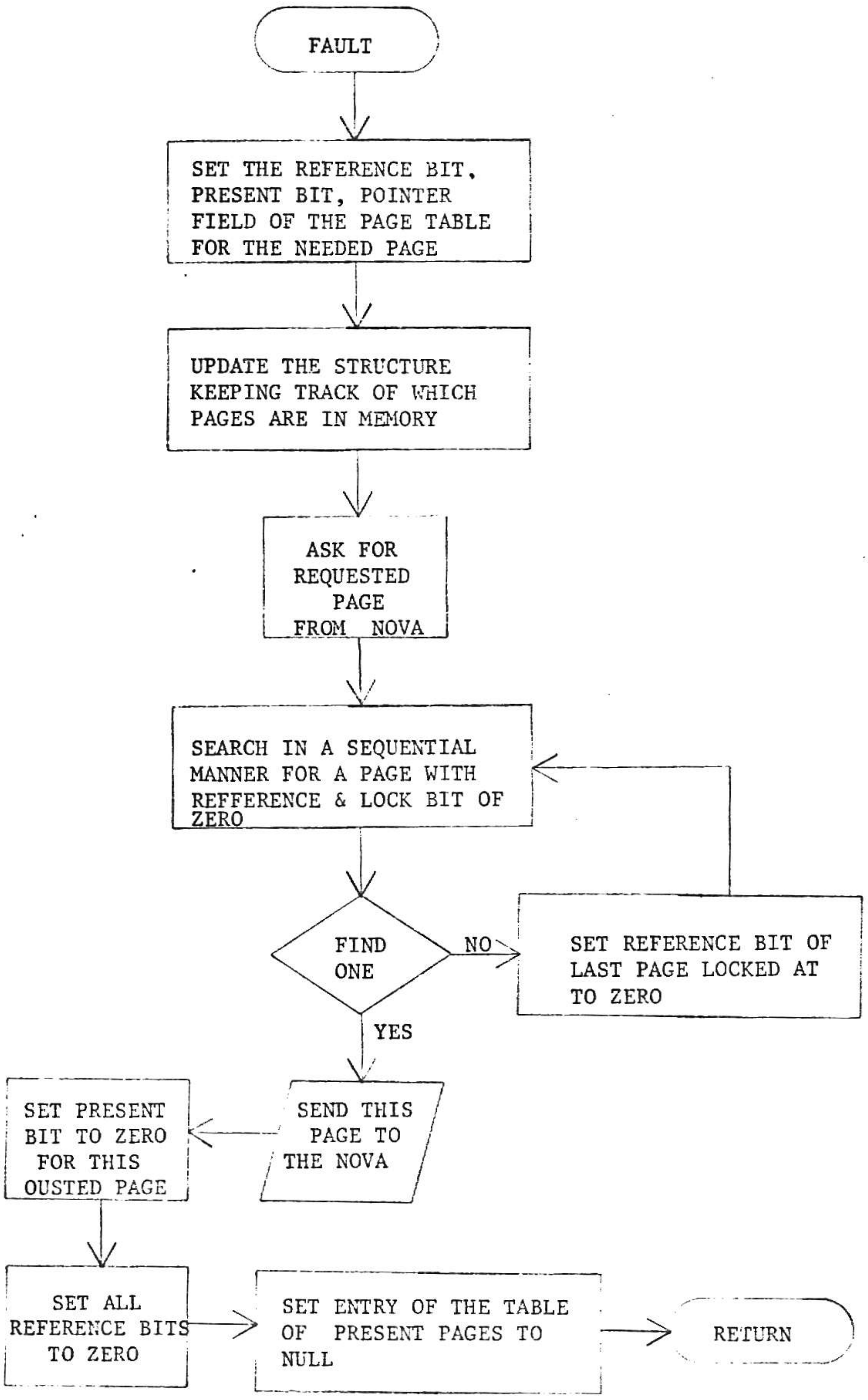
```
ATL:    A     MR1,MDR,YX,I4          START OF LIST ADDRESS IN MR1
        BAL   ADDRMAP(MR7)           REAL ADDRESS RETURNED IN MAR
        LI    MR5,'0001'             CONSTANT ONE IN MR5
        LB    MR2,MDR,NULL           MR2 HAS NUMBER OF SLOTS AVAILABLE
        LBR   MR3,MDR                MR3 HAS NUMBER OF SLOTS USED
        S     NULL,MR3,MR2           NUMBER USED-NUMBER OF SLOTS
        BALNC LISTOVF(MR7)           OVERFLOW CONDITION
        A     MDR,MDR,MR5,DW         NUMBER USED +1 PUT BACK IN
                                     NUMBER USED SLOT
        AI    MR1,MR1,'0002'         LOOK AT SECOND PARAMETER WORD
        BAL   ADDRMAP(MR7)           REAL ADDRESS IN MAR
        LB    MR4,MDR,NULL           MR4 HAS CURRENT TOP POINTER
        SX    MR4,MR4,MR5,ATL1,C     EQUAL ZERO?
        L     MR4,MR2                YES, TOP GETS LAST NODE
ATL1:   A     MR4,MR4,MR4            CALCULATE OFFSET OF SLOT FROM
                                     THE HEAD OF THE LIST
        A     MR1,MR1,MR4            ADD OFFSET TO CURRENT ADDRESS
        BAL   ADDRMAP(MR7)           MAKE IT REAL IN MAR
        A     MDR,YD,NULL,DW         STORE REGISTER IN SLOT
        NI    PSW,PSW,'FFF0'         RESET CONDITION CODES
        BAL   (MR6)(MR7)             BRANCH BACK TO THE EMULATOR
                                     CONTROL PROGRAM
```

APPENDIX D

The flowcharts for the emulator control program


EMULATOR
M1
VIRT_ADDR= INSTRUCTOR POINTER
ADDRMAP (INSTRPTR) REALADDR
IS SVC OR PRIVILEGED INSTR
YES
CALL EXTOS (REALADDR)
INSTRPTR= INSTRPTR + 1
M1
NO
IS MONITOR ON
YES
ADJUST APPROPRIATE COUNTER
NO
BRANCH TO APPROPRIATE MICROCODE ROUTINE
M1


ADDRMAP
CALCULATE VIRTUAL PAGE NUMBER
CALCULATE ADDRESS OFFSET FROM START OF PAGE
IS PAGE IN LEVEL 1 MEMORY
NO
CALL FAULT PASSING THE VIRTUAL PAGE NUMBER REQUESTED
YES
CALCULATE THE REAL ADDRESS
RETURN


FAULT
SET THE REFERENCE BIT, PRESENT BIT, POINTER FIELD OF THE PAGE TABLE FOR THE NEEDED PAGE
UPDATE THE STRUCTURE KEEPING TRACK OF WHICH PAGES ARE IN MEMORY
ASK FOR REQUESTED PAGE FROM NOVA
SEARCH IN A SEQUENTIAL MANNER FOR A PAGE WITH REFFERENCE & LOCK BIT OF ZERO
FIND ONE
NO
SET REFERENCE BIT OF LAST PAGE LOCKED AT TO ZERO
YES
SEND THIS PAGE TO THE NOVA
SET PRESENT BIT TO ZERO FOR THIS OUSTED PAGE
SET ALL REFERENCE BITS TO ZERO
SET ENTRY OF THE TABLE OF PRESENT PAGES TO NULL
RETURN

REFERENCES

(1) Madnick, Stuart E. and Donovan, John J., Operating Systems, McGraw-Hill Computer Science Series, 1974.

(2) Denning, Peter J., "Virtual Memory", ACM Computing Surveys, Vol. 2, No. 3, pp. 153-190, Sept. 1970.

(3) Stevens, Myers, and Constantine, "Structured Design", IBM Systems Journal, Vol. **, No. 2.

(4) Dijkstra, E.W. "The Structure of the T.H.E. Multiprogramming System", CACM, Vol. 11, No. 5, pp. 341-346, May 1968.

(5) Bentz, Arlan, "A Description of HIMICS's Level 2 Processor", Computer Science Department, KSU, Manhattan, Kansas.

(6) Pankhurst, R.J., "Program Overlay Techniques," CACM, Vol. 11, No. 2, pp. 119-125, Feb. 1968.

(7) Randell, B. and Kuehner, C.J., "Demand Paging in Perspective," Proceedings, AFIPS, 1968, FJCC, Vol. 33, pt. 2, pp. 1011-1018.

(8) Denning, Peter J., "On Modeling Program Behavior," Spring Joint Computer Conference, 1972.

(9) Ferrari, Domenico, "Improving Locality By Critical Working Sets," CACM, Vol. 17, No. 11, pp. 614-620, Nov. 1974.

(10) Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L., "Evaluation Techniques For Storage Hierarchies," IBM Systems Journal, Vol., No. 2., 1970.

(11) Anderson, Gary, "Hierarchical Structure," Computer Science Department, Kansas State University, Manhattan, Kansas.

(12) Katzan, H. Jr., "Storage Hierarchy Stystes," Proceedings, AFIPS, 1971, SJCC, Vol. 38, pp. 325-336.

(13) Bell, Gordon C., Newell, Allen, Computer Structures: Readings and Examples, McGraw-Hill Computer Science Series, 1971.

(14) Interdata Inc., Manual, OS/16 Multi-tasking Operating System Reference Manual, Interdata Inc. Publication Number B29-367, 1974.

HIMICS: A VIRTUAL MEMORY ENVIRONMENT FOR MINI-COMPUTERS AND A DESCRIPTION OF ITS LEVEL 1 PROCESSOR

by

DOUGLAS EUGENE SMITH

B.S., Kansas State University, 1973

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1975

The HIMICS system is a hierarchical virtual memory system for a network of mini-computers. This paper describes the design of this software system. A design for a possible physical implementation for the HIMICS system is also presented. This implementation design includes the hardware required for the system, basic algorithms needed by the system, and the data structures used in these algorithms.

The Level 1 processor used in the design is an Interdata 85 computer. The properties of this machine and its software packages which are relevant to the HIMICS system are described. The Nova computer is the Level 2 processor for the implementation design. This secondary processor is not described in this paper.

The HIMICS system will provide a virtual memory system for the network of mini-computers and also allow the emulation of high level languages. The implementation of this system should result in an increase of processor efficiency and system throughput for the mini-computers involved in the network.