A GENERAL DESIGN FOR A VIRTUAL OPERATING SYSTEM FOR THE HEWLETT-PACKARD 2100A MINICOMPUTER

by

Stephen B. Bogese, II

Thesis submitted to the Graduate Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

G.W. Gorsline, Chairman

B.C. Claybrook

F.G. Gray

May, 1974

Blacksburg, Virginia

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

The thrust of current technology in operating systems seems to have come from the large computer manufacturers, and that technology is not necessarily applicable to the minicomputer environment. As the number of minicomputers being sold is constantly increasing, more research and development needs to be spent in the design and analysis of operating systems for these machines. Therefore, this project is undertaken to provide an operating system design for a specific minicomputer, the Hewlett-Packard 2100.

The virtual type of operating system was chosen because of the substantial recent interest in and the far-ranging potential of this type of system. A key factor in the design of this operating system was the ability to implement the final design.

The design as described in this report is felt to be a sophisticated design from the viewpoint of desirable features, yet it is very practical from the implementation aspect of the design.

Chapter 1

GENERAL SYSTEM DESIGN

DESIGN OBJECTIVES

The major objectives used in developing the design of a virtual operating system were:

1. Minimization of the CPU time used to maintain system overhead.

2. Minimization of the core memory requirements of the supervisor, the system tables, and the system data areas.

3. The highest possible usage of the existing hardware and machine architecture.

4. Provide a high degree of protection for the system and for the users.

5. Specify hardware modifications and/or additions required to obtain satisfactory system performance.

SYSTEM CONFIGURATION

Virtual operating systems have been approached in two basic design configurations. These configurations are the two-level design and the three-level design. The components of the two-level design are:

1. Level one - the resident system level.
2. Level two - the virtual user level.

The components of the three-level design are:

1. Level one - the resident system level.
2. Level two - the virtual system level.
3. Level three - the virtual user level.

These two design approaches share a substantial conceptual subset, but the implementation of these distinct approaches results in very different designs. As the actual designs vary considerably, these designs were compared to determine the best suited design for this virtual operating system.

The two-level design was chosen over the three-level design based on the following considerations:

1. The relative size of the available core memory is small, with a current size of 16K words, and is expandable to a maximum size of only 32K words. If there were a virtual system level, there might be copies of the same system programs in each user's virtual area, or one copy of the system programs might be shared by all of the users. Since the amount of virtual memory associated with each task would be greater, more pages would be necessary for each task. Each additional page would require the use of additional memory for page information, as well as a page

table for each virtual system level. It is clear that the three-level design would use more memory than the two-level design, a critical consideration in a computer with a small amount of core.

2. In a three-level design, only the resident system level (level one) is core-resident. Virtual system level programs, which handle most of the supervisor work for a task, must be paged in and out as necessary. This would result in a considerable increase in paging traffic, an extremely undesirable situation, particularly since the Hewlett-Packard 2100 does not have a hard-wired paging device. The two-level design has only one level of system programs, most of which are core-resident. As a result, paging of system level programs is required only occassionally for transient routines, and the paging traffic environment is much better.

3. The Hewlett-Packard 2100 does not have an associative memory, which would have permitted dynamic page address translation to occur in parallel. This means that page address resolution will be fairly slow. The three-level design has page address translation required on two levels: (1) the virtual system level, (2) the virtual user level. As the two-level design has page address translation occurring on only one level, the virtual user level, the increase in speed of the two-level design over the three-level design is believed to be substantial. The time savings in the operation of the system results from avoiding page address translation for all virtual system level programs.

Based on the considerations described above, the two-level system configuration provides a substantial memory space and time efficiency over the three-level system configuration. This is a result of characteristics in those configurations unique to a computer with a small memory.

SYSTEM SPECIFICATIONS

The characteristics and specifications of the virtual operating system are based on (or derived from) the major objectives used to develop the system design.

The system characteristics and specifications are as follows:

1. The size of each virtual and physical page is 256 words(512 bytes).

2. Virtual page address translation is handled by a single-level mapping structure instead of a two-level mapping structure. The two-level mapping structure uses two sets of tables, segment tables and page tables. The one-level mapping structure uses only one set of tables, page tables. Because the relative size of core memory dictates a fairly small number of pages, and the limitation of the number of tasks permitted on the system concurrently establishes a probable small bound on the number of total pages on the system, the power of the two-level mapping structure for large numbers of pages is unnecessary. The one-level mapping structure is more space efficient for many active tasks since it has only one set of tables instead of two sets of tables. The one-level mapping structure, in most cases, is more suitable for the Hewlett-Packard 2100 environment.

3. Paging is handled by the two existing DMA(Direct Memory Access) channels to provide the highest possible data transmission rate for page I/O.

4. The routines comprising the supervisor will reside in the lower 4K of memory. (exceptions - refer to 6)

5. The supervisor routines belong to the resident system level and directly access memory with no page address translation. This is accomplished by the new fetch microprogram.

6. Several functions of the supervisor, the page bounds checking, virtual page address translation, and physical page address assignment (when the virtual page is core-resident), are handled in micro-code in the

new fetch microprogram.

7. The indirect addressing feature of the Hewlett-Packard 2100 forces itself on the virtual operating system because it is a hardware function uncontrollable by either software or micro-code. Since the system does not use indirect addressing for reasons of protection, the indirect microprogram must be rewritten, and the indirect bit is unusable. This creates additional changes in the micro-code. As the old jump subroutine microprogram saved the return address in the first location of each subroutine, recursion was not allowed. Returning from a subroutine was accomplished by a jump indirect through the first location of that subroutine. The jump subroutine microprogram must be rewritten, and a new microprogram for a subroutine return must be created.

8. The microprograms which must be rewritten or created are the fetch, indirect, interrupt, jump subroutine, and return microprograms.

9. The resident system level will have a system data vector(SDV) anchored at location 100_8 which contains pointers to the following system tables and vectors: the physical page vector(PPV), the system task vector (STV), the page table vector(PTV), the task control table(TCT), and the system return stack(SRS).

10. Base register addressing is provided as an integral part of each user instruction word to allow an almost unlimited number of virtual pages to be associated with each task by using only three bits to distinguish all virtual page addresses. The maximum number of virtual pages that can be provided is the difference between the amount of core used by the system and the size of protected core.

11. The types of interrupts in the virtual operating system are: power fail, parity error, page bounds, page fault, timer, page I/O, non-page I/O, and program.

12. The interrupt handling routines are supervisor routines and handle interrupts through the combined use of the hardware interrupt architecture and the system interrupt tables.

13. The system will allow a maximum of sixteen concurrent tasks including the system task.

Chapter 2

HARDWARE MODIFICATIONS

The new microprograms which would be added or would replace existing versions require that the existing versions in module zero be physically replaced, or that these new microprograms be placed in module one, two, or three, and that module zero be disabled. It is currently possible to disable module zero so that the micro-decoder logic will reference another module. This is the recommended approach.

The timer that is currently available on the Hewlett-Packard 2100 is handled through the hardware interrupt architecture as just another I/O channel. This arrangement means that the timer has a lower priority than the paging (DMA) channels, the console, and some other I/O devices. The timer is not satisfactory for time-shared applications. Two new items are needed: a new timer and a privileged interrupt board.

The privileged board is on order, and when wired in, has interrupts with higher priority than all other interrupts than presumably power fail, parity error, page bounds, and page fault. The new timer, or the old timer if necessary, should be wired into the privileged interrupt board instead of the I/O channel. This will establish the timer as consistent with the system interrupt design.

Indirect addressing is a standard feature of the Hewlett-Packard 2100. Indirect addressing is handled at the hardware level, which allows no micro-code control of this feature. Since the virtual

operating system does not use indirect addressing, the use of this indirect bit for other information is impossible because of the hardware. This characteristic of the Hewlett-Packard 2100 is undesirable because it limits the way a memory word can be used and effectively results in a loss of six percent information in the computer. To provide adequate system security, the indirect microprogram must generate an error if ever executed. It would be desirable to change this hardware feature, but no change has been specified because the cost of such a modification is not economically justifiable.

Chapter 3

ADDRESSING

INSTRUCTION WORD FORMATS

There are two distinct basic word formats for memory reference instructions. They are:

1. Virtual memory reference instructions (figure 1).
2. Resident system level memory reference instructions (figure 2).

Bits

15: 0 indicates no indirect addressing
1 indicates indirect addressing

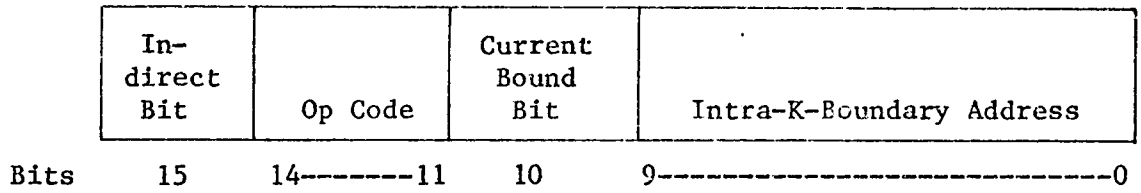
11-14: Specify the op code

10: 0 indicates no base virtual page address modification
1 indicates base virtual page address modification

8-9: 0 with bit 10 = 0 indicates that the current page is to be used as the page for the memory reference
1, 2, or 3 with bit 10 = 0 indicates that the page address is virtual page 1, 2, or 3 respectively
0, 1, 2, or 3 with bit 10 = 1 indicates that the page address is virtual page 0, 1, 2, or 3 relative to the base virtual page address

0-7: Specify the virtual intra-page address, the range of this address being from 0 to 255

Figure 1. Virtual Memory Reference Instruction



Bits

15: 0 indicates no indirect addressing
1 indicates indirect addressing

11-14: Specify the op code

10: 0 indicates that the k-boundary address zero is used for the memory reference
1 indicates that the current k-boundary address is used for the memory reference

0-9: Specify the intra-k-boundary address, the range of this address being from 0 to 1,023

Figure 2. Resident System Level Memory Reference Instruction

BASE REGISTER ADDRESSING

To allow a user task an unlimited number of virtual pages, several different methods of implementation were considered. An examination of the bit meanings of a virtual memory reference instruction shows that only two bits are used to specify a virtual page. From an informational aspect, only four states or virtual pages are distinguishable. This number of pages is insufficient to contain most programs.

There are three basic approaches to this problem. In the first approach, all virtual memory reference instructions could be two word instructions. The first word would contain the op code, and the second word would contain the virtual page address and the intra-page address. Clearly, this approach wastes over half of a memory word, and uses twice the memory space as single word instructions. This approach was eliminated as too inefficient and wasteful.

The second approach is to use the three bits (8-10) to specify the virtual page. This results in eight distinguishable states or virtual pages, likewise an insufficient number to contain most programs. The idea of using some of the intra-page address bits for virtual page address bits does not provide any benefit because increasing the number of virtual pages correspondingly reduces the number of intra-page addresses. This leaves the memory size which can be referenced unchanged.

The third approach is to utilize the concept of base register address modification. Two bits are used to specify a virtual page,

and one bit is used to indicate base virtual page address modification. The contents of the B register will be the base virtual page address, and the virtual page specified will be modified by the B register contents when base register address modification is indicated in an instruction. If the desired page is not accessible from the current base virtual page address, the contents of the base register is loaded with a new base address to make the desired page accessible. This approach is the most efficient and allows almost any number of virtual pages to be associated with one task.

There are several characteristics of this third approach which must be examined. If an instruction references a virtual page with the modification bit and that page is not accessible from the current base virtual page address, the immediately preceeding instruction must be a load base register with the needed base virtual page address. No load base register instruction is required for the following cases: (1) the virtual page referenced is the current page, (2) no base register address modification is indicated, (3) the virtual page is modified and accessible from the current base virtual page address. These characteristics will place considerable burdens on the compilers, assemblers, and linkage editors creating modules for this system. The characteristics of this approach have implications in the area of system security. Since address modification by a register is permitted, the fetch microprogram must calculate the complete virtual page address prior to performing virtual page address translation to ensure that all memory references are adequately checked for validity.

LOCALITY OF ADDRESSING

The concept of the locality of addressing has been extended with reference to this virtual operating system.

In previous work on the locality of addressing, locality is almost always examined with respect to modules of pure code. The extension of this concept is with respect to modules of pure data.

As more address locality occurs in memory references to pure code, the proportion of references to the current virtual page from that same page will increase. This increase will result in a corresponding savings in execution time as no virtual page translation takes place in references from a virtual page to itself. This time savings is substantial as the amount of micro-code needed to handle current page references is about one-third of the amount of code needed for virtual page address translation.

The locality of addressing to virtual data pages will achieve both memory space and time savings. As the memory references have more locality, fewer load base register instructions are needed for any string of memory references. Additionally, as the locality of addressing to global variables increases, not only are the number of load base register instructions reduced, but even fewer memory references must be modified by a base register address.

Chapter 4

REQUIREMENTS FOR COMPILERS, ASSEMBLERS, AND LINKAGE EDITORS

The use of base register address modification for virtual page addresses means that compilers and other translator programs as well as the user can generate the load base register instruction. Thus, the contents of the base register can and will be modified dynamically during program execution. To make the programs on the system as space and time efficient as possible, several minimization techniques were developed by the author to reduce the number of load base register instructions created by the compilers or other translator programs.

The minimization techniques developed to reduce the number of load base register instructions are the following:

1. Place all global data areas in non-base register modified pages.

2. Place as many local data areas as possible in non-base register modified pages.

3. Place all local data areas into intra-page locations sequentially and fill pages in the order of their virtual page addresses. This will cause data pages to have contiguous virtual page addresses.

4. Create a special page designator that can be used by any page. This designator will indicate a memory reference from a page to itself.

For the minimization techniques to be effective, the following assumptions were made:

1. Seventy-five percent of all memory references in a program is to the program's data.

2. The remaining twenty-five percent of all memory references are divided between three groups:

    a. References to memory addresses within the current virtual page.

    b. References to memory addresses accessible from the current base virtual page address.

    c. References to memory addresses not accessible from the current base virtual page address.

3. All program labels or entry points that are never referenced are extraneous and should not generate any load base register instructions.

The compiler or other translator programs must perform the following functions when translating a program:

1. The program code and the program data must be partitioned into virtual code pages and virtual data pages respectively.

2. The virtual data pages are constructed from all memory locations in the program used to hold data. These virtual data pages are constructed using the following procedure:

    a. Collect all global variables and assign them to the three non-modified virtual pages in order until all have been assigned. If additional pages are needed, use the modified virtual pages in order until all global variables have been assigned.

    b. Collect all local variables and assign them to any remaining space in the non-modified virtual pages in order, then to the modified virtual pages in order until all local variables have been assigned.

    c. The only restriction on this procedure is that the first thirty-three locations in non-modified virtual page one may not be used to hold any data, as those locations are used as a run-time subroutine stack.

3. The virtual code pages are constructed from the program code using the first available virtual page after

the virtual data pages have been constructed.

4. At every label or entry point in the program code, a NOP must be inserted at those points, and the original statement associated with that entry point is made the next sequential statement and is unlabeled.

5. The indirect bit in every instruction word is assigned a zero.

The functions described above have several important implications. Since virtual pages are either virtual data pages or virtual code pages, the virtual code pages consist of pure code. No modification of pure code is likely to occur, and as a result the virtual code pages should rarely be changed. This will result in a much better page traffic environment since only the virtual data pages need be rewritten onto the paging device.

As most or all of the global variables will reside on the non-modified virtual pages, no load base register instructions will be needed for memory references to global variables. As the number of global variables increase up to a certain bound, the number of load base register instructions will decrease.

The effect of these functions is to minimize the number of load base register instructions and improve the time and space efficiency of a program running on the system.

The linkage editor must perform the following functions on modules it receives:

1. Every external entry point or label is checked through the external symbol dictionary to determine whether it is ever referenced. All points not referenced are flagged.

2. For each memory reference, the virtual page address desired is checked to see if it is accessible directly (the non-base register modified pages), or if that page is accessible using the current base virtual page address. If it is accessible, the NOP is not replaced by a load base register instruction. If it is not accessible, the loan base register instruction replaces the NOP. All cross-references to entry points are checked, and if that entry point is referenced by instructions using different base virtual page addresses, the load base register instruction replaces the NOP.

The use of NOP's where no load base register instruction is needed further minimizes the number of those load instructions.

More minimization is attainable through techniques of program writing such as locality of addressing.

Chapter 5

INPUT/OUTPUT

TYPES OF INPUT/OUTPUT

There are two types of input/output (I/O) under the virtual operating system. The two types are page I/O and non-page I/O.

Page I/O is designed to read in and write out pages associated with a task on the system. This I/O is handled by the two existing DMA channels on the HP 2100, and the paging devices will be the two disks on the HP 2100. The interrupt priority of the two DMA channels is higher than all other channels, but lower than the power fail, parity error, page bounds, page fault, and timer interrupt priorities.

Non-page I/O is designed to read in and write out any non-page data. Each I/O channel has a unique hardware priority associated with it. The operators console has the highest non-page I/O priority, followed by the other I/O devices with the exception of the timer that is wired to the privileged interrupt board.

PAGING ALGORITHMS

No paging algorithm has been defined for this virtual operating system. However, the author intuitively feels that the LRU (least recently used) page replacement algorithms with certain modifications may be more desirable than other types of replacement algorithms for this operating system. The changes in the LRU algorithms would involve the replacement status of data pages, and the importance of keeping non-modified virtual page one core-resident. Virtual page one has a two-fold importance: (a) it contains the task's subroutine return stack, (b) it contains global data areas. The new fetch micro-program is designed to work in a demand paging environment, but it is not limited to only that environment.

The study of paging and its associated algorithms is a complex topic and could be a project in itself. Thus, this area of study was not undertaken.

Chapter 6

SYSTEM PROTECTION

The virtual operating system has been designed with three basic categories of protection. Those categories are:

1. Virtual address translation for all user programs.
2. Establishment of a protected core memory area.
3. The creation of a set of privileged instructions.

The first category has two important functions, virtual page address bounds checking and virtual page address mapping into a subset of all possible physical page addresses. Both of these functions are handled in the new fetch micro-program to attain the fastest possible execution of these functions and ensure protection from the user programs. Since any virtual page address is modifiable by a base virtual page address, the fetch micro-program places the base address in a register, and if modification is indicated, adds it to the virtual page specified, and generates the complete virtual page address prior to the bounds checking or the page mapping. This prevents a user from overlaying the address created by the linkage editor in a load base register instruction in an attempt to access a physical page not associated with his task. The complete virtual page address is checked to see if it exists within the active task's page table. If that virtual page is valid, then a check is made to determine if that page is core-resident. If so, the physical page address is combined with the intra-page address, saved, and the fetch micro-program terminates. If the virtual

page is valid but not core-resident, an interrupt is set to indicate a page fault. If the virtual page is not valid, an interrupt is set to indicate a page bound exception. The fetch micro-program then terminates.

The second category not only establishes a protected memory area, but also places the system tables and vectors in that area. The system area is established at system load time by loading the F (fence) register with the upper boundary of the system area. All supervisor programs and system tables and vectors will reside in this protected memory area. The supervisor never places any user virtual page in this memory area or places a physical page address in a page table which is in this protected area. Since every user memory reference is mapped through his page table by the fetch micro-program, it is impossible for a user task to modify any system tables or any page tables as their addresses are not in any user's page table.

In order to make the supervisor as time-efficient as possible, supervisor tasks reference physical memory locations directly without any address mapping. As a result, the fetch micro-program must be able to determine if a given task is a supervisor task or a user task. This is accomplished by loading the F register with the upper boundary of protected core at system load time. The F register is then compared with the P register. If the contects of the P register is less than the contents of the F register, the task is a supervisor task and no virtual address translation occurs. Otherwise, the task is a user task and virtual address translation takes place. Therefore, any task which can

execute in the lower protected core is regarded as a supervisor task and can do anything.

The third category creates a set of privileged instructions in order to prevent certain instructions from being used by a user task to access lower protected core. The two types of instructions which are being made privileged are:

1. All I/O instructions.
2. All instructions which can affect the interrupt status of the HP 2100.

The instructions of type one are OTA and OTB. The instructions of type two are STF, CLF, STC, and CLC. These instructions will be made privileged by adding new micro-code routine to compare the F register to the P register. If the P register contents are larger, the current task is a user task and NOP's will be executed. If the F register contents are larger, the task is a supervisor task and the micro-program is executed. This will prevent a user task from disabling the memory protect interrupt logic used to detect page bounds exceptions and page faults in order to access protected core.

The protection designed for this system is believed to be complete and substantial. It is also intended to be an open-ended design, allowing for future modification or changes. This substantial protection described is accomplished by a tradeoff of two of the major design objectives of the system, minimization of CPU time and system protection. Although this protection was achieved by a sacrifice of the time efficiency of the virtual operating system, this tradeoff is felt

to be the best tradeoff in terms of overall system objectives.

Chapter 7

INTERRUPTS

The virtual operating system handles interrupts as a combination of hardware and software. This arrangement permits each component to handle the aspects of interrupts for which it is best suited. The hardware interrupt architecture is a priority linked network of gates.

With the timer rewired from an I/O channel to the privileged interrupt board, the priority order, select code, and interrupt type are shown below:

Priority	Select Code	Interrupt Type
1	04	power fail
2	05	parity error/page bounds/page fault
3		timer
4	06	DMA channel 1 (page I/O)
5	07	DMA channel 2 (page I/O)
6	10	I/O channel (non-page I/O)
73	77	I/O channel (non-page I/O)

When an interrupt which has occurred is accepted, a jump subroutine instruction in the location corresponding to the select code will be executed. These jumps will be to the appropriate part of the interrupt stacker for that type interrupt, with the exception of power fail and parity error interrupts. A power fail interrupt causes a jump to a system save routine, while a parity error interrupt may cause a jump to a retry routine. As the interrupt stacker adds the pending interrupt to the interrupt table, the existence of that type of interrupt is marked in the system interrupt table (SIT). As the stacker

adds this interrupt to the interrupt table, all interrupts will be disabled until all necessary data has been stored, after which interrupts will again be accepted. When no more hardware interrupts occur, the stacker will turn control over to the interrupt scanner. Then the interrupt scanner will find the highest priority interrupt in the interrupt table and execute a jump subroutine to that interrupt handler. If an interrupt handler needs to create an entry in the interrupt table, it calls the stacker with the appropriate information and sets a hardware flag. When software created interrupts are stacked, they are checked by the scanner, and interrupts are handled in priority order. Although there are two different hardware priorities for page I/O interrupts, and many different hardware priorities for non-page I/O interrupts, all page I/O interrupts have the same software priority. All non-page I/O interrupts have the same software priority.

The system interrupt routines do not modify the hardware interrupt flags established by the interrupts until the appropriate interrupt handler has taken care of that interrupt. Each interrupt handler then clears the interrupt flag as its last function. If higher priority interrupts occur during the operation of the interrupt stacker, they cannot interrupt the stacker because they will be masked off. At any other time, such as during the execution of an interrupt handler, a hardware interrupt of higher priority than that being serviced will automatically cause control to be turned over to the appropriate interrupt stacker. After that higher priority interrupt has been serviced, control will be returned to the interrupt handler previously

executing.

The arrangement described above uses the hardware interrupt architecture to respond to the highest priority interrupt on the system, and since the highest priority interrupt hardware disables all lower priority interrupts, response time to service the highest priority interrupt pending is minimized. This is because the interrupt stacker will not be stacking lower priority interrupts, but it will have the appropriate interrupt handler working on that high priority interrupt. As the high priority interrupts are completed, their interrupt flags are cleared, and lower priority interrupts can capture the interrupt stacker.

The four basic software interrupts in the system interrupt table in order of priority are:

1. Timer.
2. Page I/O.
3. Non-page I/O.
4. Program.

The software interrupt information needed by the system is contained in the system interrupt table. Interrupt handling by the software is accomplished by a first in - first out algorithm for interrupts of equal priority, with the highest priority interrupts serviced first.

The system interrupt design is believed to provide the fastest and most efficient interrupt capabilities for the virtual operating system using the existing hardware architecture with only very slight modifications.

Chapter 8

SYSTEM MICROPROGRAMS

NEW MICROPROGRAMS

The new microprograms needed for the virtual operating system are the fetch, indirect, interrupt, jump subroutine, and return.

There are many other microprograms which must be modified, but the five mentioned above are characterized as new because each must be completely rewritten. The fetch microprogram is discussed under the system protection section, and a sample fetch microprogram will follow. The new fetch microprogram will be considerably slower than the existing fetch. The indirect microprogram must be able to set an interrupt, as indirect addressing is no longer permitted. The interrupt microprogram must be expanded to include the ability to check and recognize certain types of interrupts.

The jump subroutine and the return microprograms are closely related. Subroutine jumps will be handled in stacks. There will be a system return stack (SRS) for the physical system level programs, and each task will have its return stack in the first thirty-three locations of non-modified virtual page 1. The jump subroutine stores the return address in the next available location in the stack by first updating the number of entries and then uses that information to store the return address. By using the F register to determine if the active task is a supervisor task, the jump subroutine decides whether to store

the return address in the system return stack. The system return stack contains physical addresses. If the active task is a user task, then virtual page one is looked up and the virtual page return address is stored by updating the number of entries and then using that to store the return address.

The return will execute a return to the next lower level in the sequence of called routines. For system level programs, the number of entries is accessed and used to obtain the physical return address, after which the number of entries is decremented and stored. The user task will cause the microprogram to access the number of entries, then use that number to obtain the virtual return address, translate the virtual address into the corresponding physical address and execute the branch to that address. Although this means that the user has access to his task return stack, the system protection is effective because virtual page address translation takes place before any branching occurs. By using this implementation, the user who wishes to return several levels at once need only change the number of entries word in order to accomplish this. It is the user's responsibility to correctly establish the contents of the base register for the intended return level when using a multiple level return. The implication of this feature is that locality of addressing properties may be adversely affected.

The use of a return stack has two desirable features:

1. Multiple level returns can easily be controlled and set to occur.

2. Recursion is allowed up to a certain bound. The original HP 2100 return scheme did not allow recursion. Thus, the return stack results in a major system design improvement.

FETCH MICROPROGRAM

	R	S	FN	ST	SP	SK	
		P	IOR	M	RW		
		T	IOR	Q			
	Q		IOR	IR			
	F	P	DEC		RSS	NEG	checks for system task
			JMP		USER		if so, skips branch to user code
		ADR	CFLG	S1		EOP	
		CR	IOR	Q	0000	0000	
USER	B		IOR	S4			save base register contents
		CL	IOR	B	0000	0111	
	Q		IOR	S2			
	B	S2	AND			NAAB	see if reference is to current page
			JMP		MAP		if so, skip branch to mapping code
		CL	IOR	A	1111	1111	
	A	P	AND	A			
	A	ADR	IOR	S1			
		S4	IOR	B		EOP	restores base register
		CR	IOR	Q	0000	0000	

	R	S	FN	ST	SP	SK	
MAP		CR	IOR	B	0100	0001	
	B		IOR	M	RW		access to location 101_8
		T	DEC	M	RW		for STV + 1 for active task offset word
		T	IOR	A			
	A	CR	AND	A	0011	1111	mask obtains exact active
	B		INC	M	RW		task offset access to lo-
	A	T	ADD	M	RW		cation 102_8 for PTV add task offset to base location of PTV, access page table address
		T	IOR	S2	RSS	NEG	check for valid in-core
			JMP		NEPT		page table address, if valid, skip branch to error handler
		S2	IOR	M	RW		access base location of page table for no. of pages
		T	IOR	B			save no. of pages
		CL	IOR	A	0000	0011	
	A	ADR	AND	A	R1		use mask to isolate virtual page address
	A		IOR	A	R1		

	R	S	FN	ST	SP	SK	
	A		IOR	A	R1		shift virtual page address to obtain right-justified value
	A		IOR	A	R1		
	A		IOR	A	R1		
	A		IOR	A	R1		
	A		IOR	A	R1		
	A		IOR	S3	R1		save right-justified address
	Q	CL	AND		0000	0100	use mask to see if base register address modification is indicated, if so, skip branch to no base register modify code
			IOR		RSS	NAAB	
			JMP		NOBRM		
		S4	IOR	A			S4 has saved base register contents
	A	S3	ADD	S1		UNC	modify virtual page address, save and skip
NOBRM		S3	IOR	S1			save unmodified virtual page address
	B	S1	DEC		RSS	NEG	if virtual page address is less than no. of pages, skip branch to page bounds code
			JMP		PBXCPT		
		S1	DEC		RSS	NEC	if virtual page address is greater than 1, skip branch to
			JMP		PBXCPT		

	R	S	FN	ST	SP	SK	
							page bounds code
		S1	IOR	A	L1		left shift virtual page address by 1, to obtain double-word alignment
	A	S2	ADD	M	RW		add base location of page table to aligned virtual page address and access
		M	INC	S2			save address of DASD page address word
		T	IOR	S3		ODD	check valid core address bit, if valid, skip branch to page fault code
			JMP		PGFLT		
		CL	IOR	A	1111	1111	
	A	S3	AND	A			use mask to obtain physical page address
		CR	IOR	B	1111	1111	
	B	ADR	AND	S3			use mask to obtain intra-page address from instruction word
	A	S3	IOR	S1			create and save complete physical address
		S4	IOR	B		EOP	restore base register
		CR	IOR	Q	0000	0000	
NEPT		CL	IOR	S1	1111	1111	

	R	S	FN	ST	SF	SK	
	F	S1	DEC			NMPV	generate signal to memory protect logic
		S4	IOR	B		EOP	restore base register
		CR	IOR	Q	0000	0100	store interrupt type in Q-register
PBXCPT		CL	IOR	S1	1111	1111	
	F	S1	DEC			NMPV	generate signal to memory protect logic
		S4	IOR	B		EOP	restore base register
		CR	IOR	Q	0000	0010	store interrupt type in Q-register
PGFLT		CL	IOR	S1	1111	1111	
	F	S1	DEC			NMPV	generate signal to memory protect logic
		S4	IOR	B			restore base register
		S2	IOR	A			save address of word containing DASD page address
		CR	IOR	Q	0000	0001	store interrupt type in Q-register

Chapter 9

SYSTEM TABLES, VECTORS, AND STACKS

The major tables and vectors used by the system are:

1. System Data Vector (SDV)
2. Physical Page Vector (PPV)
3. System Task Vector (STV)
4. Page Table Vector (PTV)
5. Page Tables
6. Task Control Table (TCT)
7. Task Save Table (TST)
8. System Interrupt Table (SIT)
9. System Return Stack (SRS)

The System Data Vector is anchored in core at location 100_8. It contains pointers to the most important system tables and vectors (figure 3).

The Physical Page Vector contains information on each physical page in the form of control bits, and the number of available pages. Each available page is part of a linked list, with a special entry containing a pointer to the head of the linked list (figure 4).

The System Task Vector is a table containing a doubleword entry for each task on the system. There are five pointer links to several linked lists. The links are:

1. To the top of the dispatchable task list.
2. To the top of the eligible task list (that entry will

contain a link to the bottom of the dispatchable task list).

3. To the bottom of the eligible task list.
4. To the top of the inactive task list.
5. To the bottom of the inactive task list.

The table also contains the active task offset. Each task entry contains links to the preceding and succeeding tasks in that list. Each entry also contains an offset to the PTV (figure 5).

The Page Table Vector is a vector which contains the core address of a task's page table, a core address validity bit, which indicates whether that address is valid/invalid, and a direct-access device address of that page table (figure 6).

A Page Table exists for each task in the system. Each table contains the length of the table, the table control bits, the physical page address for each virtual page (if in core), the core address validity bit, the page change bit, and a direct-access device address for the page (figure 7).

The Task Control Table contains a four-word entry for each task with the following information: the user id, the task priority, an interrupt mask, task flags, the input and output device address for the task, the TST core address, a TST core address validity bit, and a direct-access device address for the TST (figure 8).

The Task Save Table contains save areas for the P, M, T, A, B, Q, and F registers, the AA, BA, flag, extend, and overflow flip-flops; the counter, a time-slice bit, and the remaining time in the time slice.

A pointer back to the task's TCT is also contained (figure 9). There is one TST for each task.

The System Interrupt Table contains pointers to the first and last ISIE (Individual System Interrupt Entry) in the linked list for each type of interrupt (timer, page I/O, non-page I/O, program). The address for each interrupt handler is also in the table, as well as an interrupt mask for each type of interrupt. There is also an interrupt flag for each type of interrupt, which indicates whether there are any ISIE's in that interrupt linked list. This flag provides a fast test to determine whether or not there are any interrupts of a certain type pending (figure 10).

The Individual System Interrupt Entry contains links to the preceding and succeeding ISIE's in that interrupt linked list, status flags, an offset to the TCT, and interrupt mask associated with this individual interrupt, and an extended ISIE bit (figure 11). The TCT offset identified which task the interrupt is associated with, while the extended ISIE bit indicates that the ISIE is an additional word in length. This word may contain a pointer to an I/O Block (IOB), a Page I/O Block (PIOB), or a Message Control Block (MCB).

The System Return Stack contains information on return addresses for nested calls. The first entry is the number of return addresses in the stack, and the remaining entries are the physical return addresses in the order of their calls (figure 12).

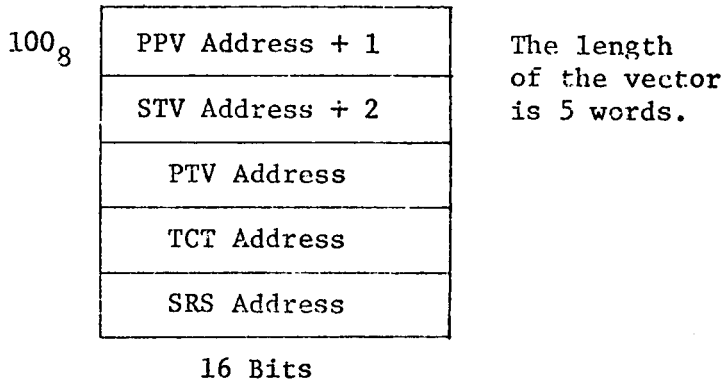


Figure 3. System Data Vector (SDV)

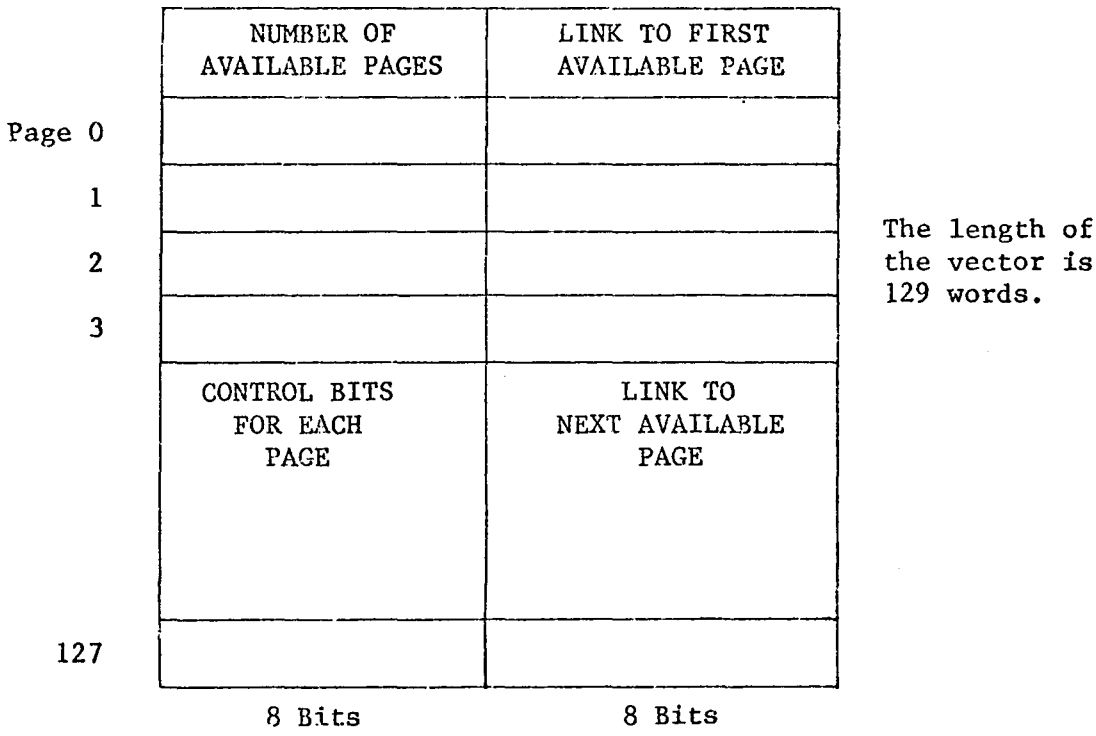


Figure 4. Physical Page Vector (PPV)

	LINK TO THE TOP OF THE DISPATCHABLE LIST	LINK TO THE TOP OF THE ELIGIBLE LIST	LINK TO THE BOTTOM OF THE ELIGIBLE LIST
	LINK TO THE TOP OF THE INACTIVE LIST	LINK TO THE BOTTOM OF THE INACTIVE LIST	ACTIVE TASK OFFSET
Task 1	LINK TO THE PRECEDING TASK	LINK TO THE SUCCEEDING TASK	OFFSET TO THE PTV
2			
3			
16			
	5 Bits	5 Bits	6 Bits

The length of the vector is 18 words.

Figure 5. System Task Vector (STV)

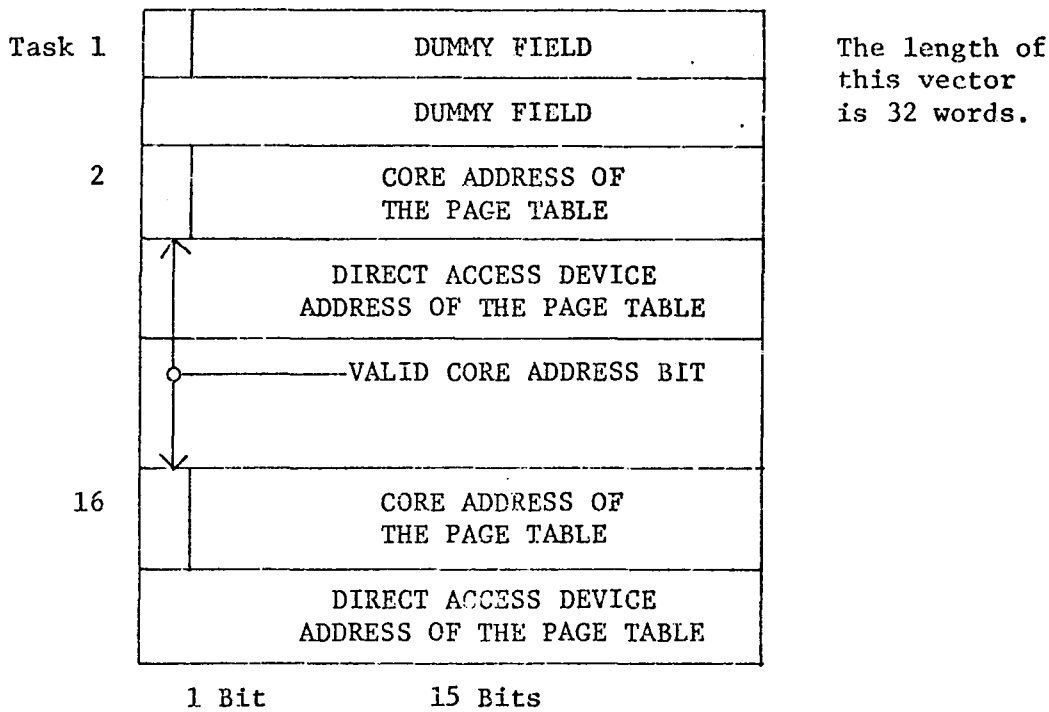


Figure 6. Page Table Vector (PTV)

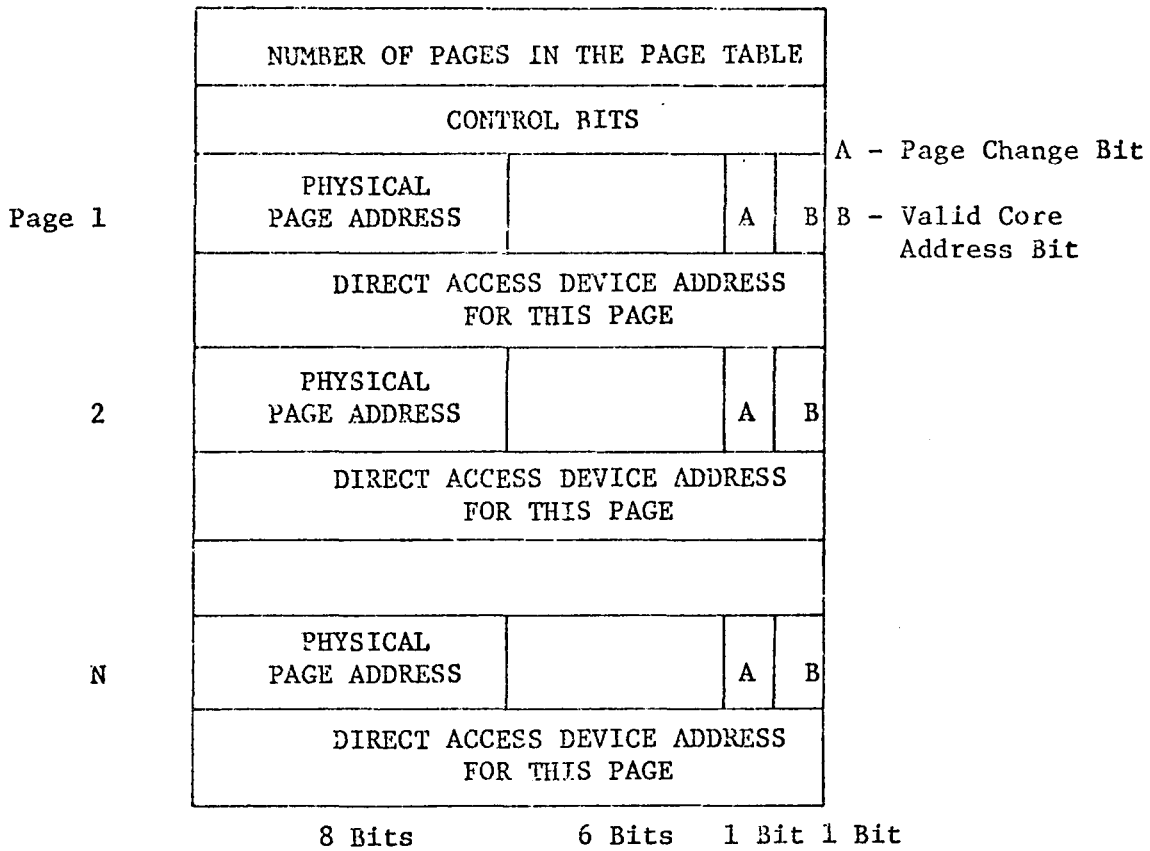
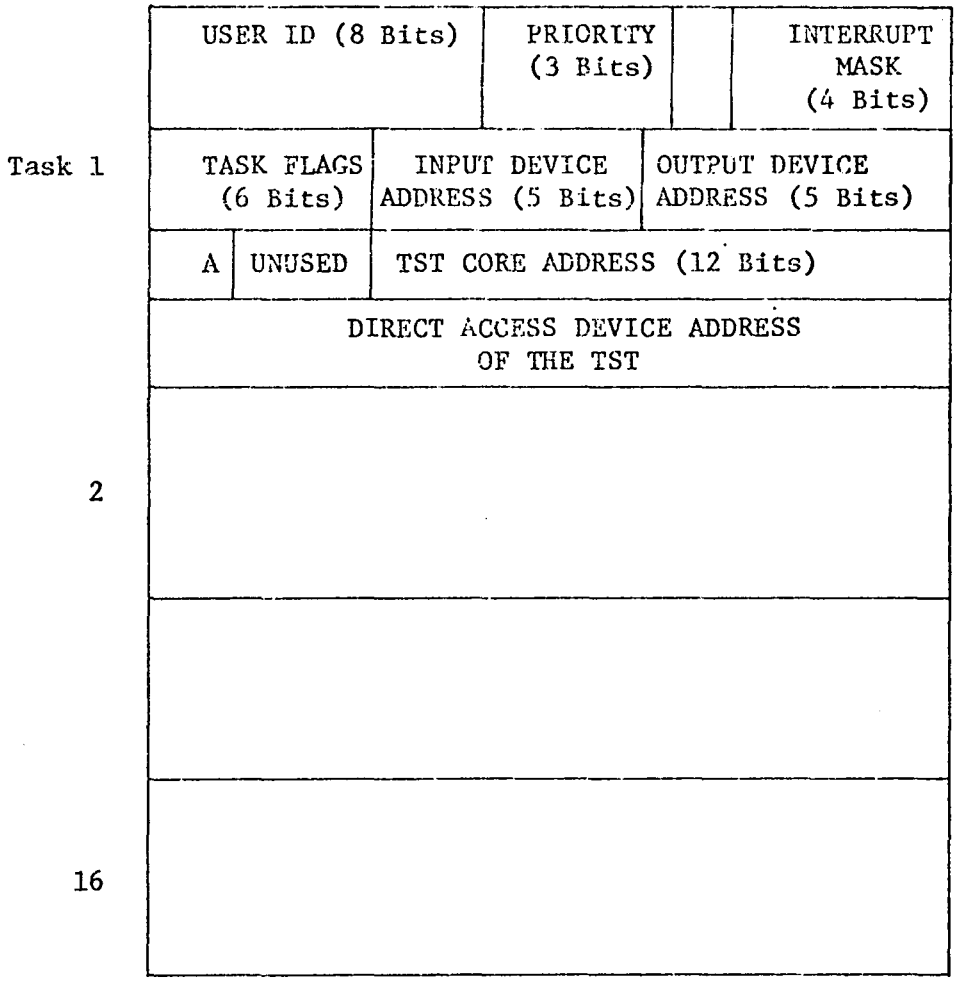


Figure 7. Page Table

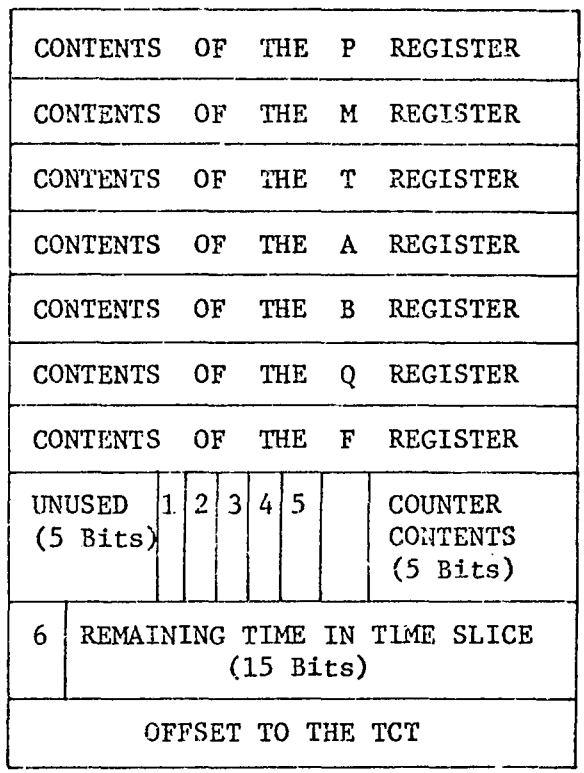


16 Bits

TASK FLAGS - Dispatchable, Eligible (1 Bit)
Active, Inactive (1 Bit)

INTERRUPT PENDING: Timer (1 Bit), Page I/O (1 Bit), Non-Page I/O (1 Bit), Program (1 Bit)

Figure 8. Task Control Table (TCT)

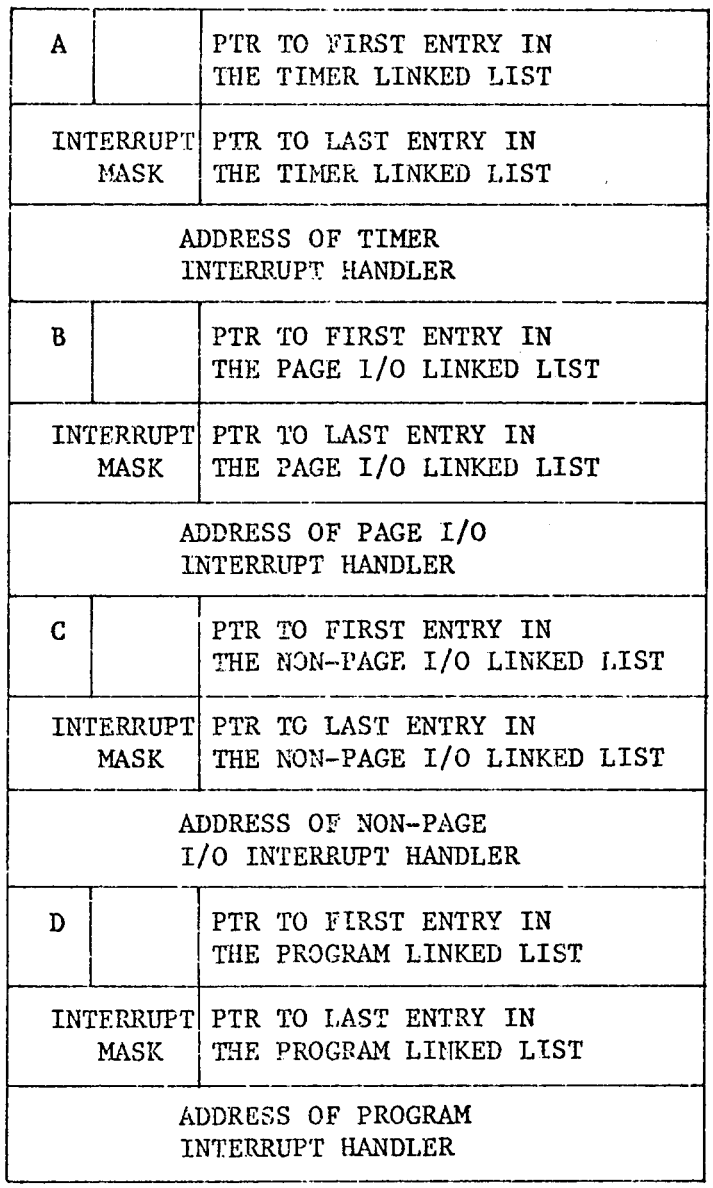


The length of this table is 10 words.

Contents of:

1. AA FF (1 Bit)
2. BA FF (1 Bit)
3. FLAG FF (1 Bit)
4. EXTEND FF (1 Bit)
5. OVERFLOW FF (1 Bit)
6. TIME SLICE IN EFFECT/ NOT IN EFFECT (1 Bit)

Figure 9. Task Save Table (TST)



The length of this table is 12 words.

A - Timer Interrupt Flag (1 Bit)

B - Page I/O Interrupt Flag (1 Bit)

C - Non-Page I/O Interrupt Flag (1 Bit)

D - Program Interrupt Flag (1 Bit)

Figure 10. System Interrupt Table (SIT)

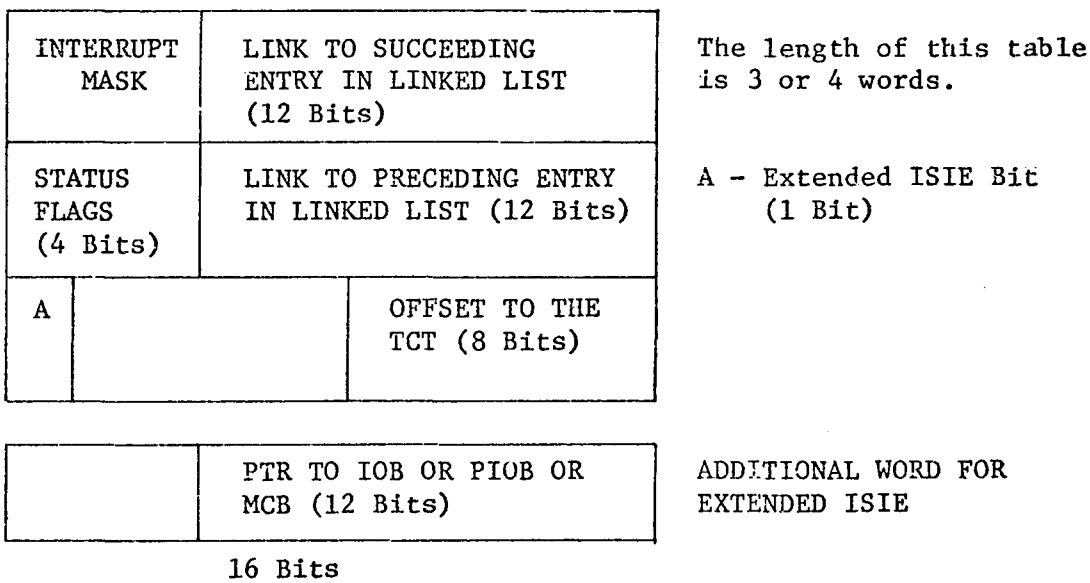
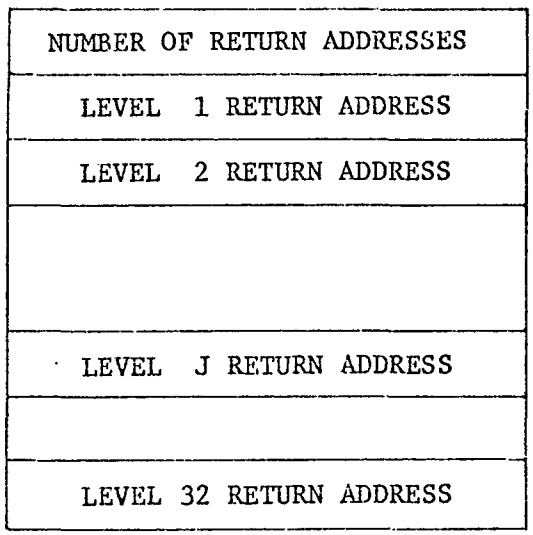


Figure 11. Individual System Interrupt Entry (ISIE)



The length of this stack is 33 words.

Figure 12. System Return Stack (SRS)

LITERATURE CITED

1. Hewlett-Packard. Model 2100A Computer Diagrams Manual. Hewlett-Packard, 1972.

2. Hewlett-Packard. Model 2100A Computer Installation and Maintenance Manual. Hewlett-Packard, 1973.

3. Hewlett-Packard. A Pocket Guide to the Hewlett-Packard 2100A Computer. Hewlett-Packard, 1972.

4. Hewlett-Packard. Microprogramming Guide for Hewlett-Packard Model 2100 Computer. Hewlett-Packard, 1972.

5. Hewlett-Packard. Microprogramming Software for Hewlett-Packard Model 2100 Computer. Hewlett-Packard, 1973.

6. Denning, P. J. "Virtual Memory." Computing Surveys 2, 3 (September, 1970), 153-189.

7. Knuth, Donald E. An Empirical Study of Fortran Programs. Computer Science Department Report, Stanford University, Calif., 1970. Springfield, Va.: Clearinghouse for Federal Scientific and Technical Information.

8. Katzan, Harry Jr. Operating Systems: A Pragmatic Approach. New York: Van Nostrand Reinhold Company, 1973.

9. Coffman, Edward G. Jr., and Denning, Peter J. Operating Systems Theory. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

The vita has been removed from the scanned document

A GENERAL DESIGN FOR A VIRTUAL OPERATING SYSTEM FOR THE HEWLETT-PACKARD 2100A MINICOMPUTER

by

Stephen B. Bogese II

(ABSTRACT)

This project presents a design for a virtual operating system for the Hewlett-Packard 2100A minicomputer.

The design was developed from the characteristics desirable in an operating system and from the restrictions imposed by the computer hardware.

The characteristics and the method of operation are specified for the major components of the operating system. The major system tables are also specified in detail.

The final design is capable of being implemented on this minicomputer.