Short title:- MULTIPROGRAMMING OF IBM 360
COMPUTER OPERATING SYSTEMS

AUTHOR: Alan Greenberg

TITLE: A HARDWARE-SOFTWARE TECHNIQUE FOR MULTIPROGRAMMING TWO UNRELATED OPERATING SYSTEMS ON AN IBM 360 COMPUTER

DEPARTMENT: Computer Science

DEGREE: Master of Science

A B S T R A C T

A method of multiprogramming two discrete operating systems simultaneously on one IBM System/360 computer is presented. This hardware-software combination is called a HYPERVISOR.

The growth of computer technology which led to the need for such a system is traced. A variety of solutions are proposed. The paper then presents one of these, the Hypervisor, from its initial planning stages, through to its implementation and testing. An evaluation of the Hypervisor is made. A discussion of the future of the Hypervisor concludes the paper.

AUTHOR: Alan Greenberg

TITLE: A HARDWARE-SOFTWARE TECHNIQUE FOR MULTIPROGRAMMING TWO UNRELATED OPERATING SYSTEMS ON AN IBM 360 COMPUTER

DEPARTMENT: Computer Science

DEGREE: Master of Science

A B R E G E

Cet ouvrage présente une méthode de multiprogrammation utilisant deux systèmes d'exploitation distincts simultanément sur un même ordinateur IBM/360. Cette combinaison d'équipement et de programmerie s'appelle l'HYPERVISOR.

L'évolution rapide des techniques de l'ordinateur conduisant vers cette combinaison est décrite. Une variété de solutions y sont proposées. Puis les phases de planification, de développement et de mise en marche sont présentées en detail pour une solution spécifique, nommément l'HYPERVISOR. La thèse se termine par une évaluation de la solution décrite et une discussion de son application future.

A HARDWARE-SOFTWARE TECHNIQUE FOR MULTIPROGRAMMING TWO UNRELATED OPERATING SYSTEMS ON AN IBM 360 COMPUTER

Master of Science Thesis

Alan Greenberg

COMPUTER SCIENCE
McGILL UNIVERSITY
MONTREAL, CANADA

AUGUST, 1973

PREFACE

A basic requirement in the management of a large-scale computing facility is to provide the complete range of services which the user community requires. Often, even if the desired facilities could be obtained, it is impractical due to cost, machine configuration, or incompatibility with existing software.

This paper describes the development of a means by which two discrete operating systems, each of which normally operates on a dedicated IBM S/360 or S/370 computer, can be multiprogrammed to operate on the same machine. Results of the project are described, and future implementations are discussed.

The reader is assumed to have some knowledge of modern digital computer internals, but not necessarily of IBM computers. Chapter 3 includes a description of all the features of S/360 architecture which are required by the rest of the paper.

The author wishes to express his thanks to Professor W.D. Thorpe, Director and Professor A.M. Valenti, Associate Director of the McGill University Computing

Centre, for their encouragement and confidence throughout the development of both the McGill-RAX System and the Hypervisor described in this paper.

Special thanks are due to Mr. Roy Miller, who worked together with the author during the past several years developing the McGill-RAX Time-sharing System, which formed the base for the Hypervisor project. The assistance of Mr. Peter Mann, the IBM representative at McGill, in satisfying the author's many requests for information is greatly appreciated.

The project to be described was carried out by the McGill University Computing Centre, under contract to The Illinois Bell Telephone Company, Chicago, Illinois. Without their complete faith and confidence in the McGill Computing Centre, this whole project would never have existed. Specific thanks are due to Madelon Clymo, Ken Hahn and Janie Melzor of Illinois Bell.

This thesis was prepared using the Administrative Terminal System (ATS) running on the IBM S/360 Model 75 computer at the McGill University Computing Centre.

TABLE OF CONTENTS

Chapter

1. COMPUTER SYSTEM DEVELOPMENT 1
2. ILLINOIS BELL TELEPHONE 7
3. SYSTEM/360 COMPUTER ARCHITECTURE. 16
4. McGILL-RAX - OS/360 HYPERVISOR DESIGN 29
5. HYPERVISOR IMPLEMENTATION 35
6. HYPERVISOR INTERNAL DESIGN AND OPERATION. . . 41
7. OPERATING SYSTEM MODIFICATIONS. 54
8. HYPERVISOR PERFORMANCE. 61
9. OVERALL RESULTS AND THE FUTURE OF THE HYPERVISOR. 73
10. CONCLUSIONS 79

APPENDIX A 80

BIBLIOGRAPHY 91

CHAPTER 1

COMPUTER SYSTEMS DEVELOPMENT

The methods by which large scale digital computers are operated have changed drastically in the last fifteen years.[1] The first computers were relatively slow and little support programming was available. To solve a problem, the user would have to reserve the machine for his exclusive use, and debug and run his job at the computer console. Although this was not a very satisfactory approach, since the machine was idle during the programmer's "think time", it was still an acceptable method of operation. System time, even on a large configuration, was often worth only twenty five to fifty dollars per hour.

As machine speed and cost increased, one man sitting at the computer console could not in general keep the machine busy even a small percentage of the time. Obviously, for a computer worth perhaps four hundred dollars per hour, some other method of operation had to be devised.

The initial solution involved several new developments. The first of these was the introduction of HIGH LEVEL LANGUAGES. A high level language such as FORTRAN or COBOL enables the computer user to present the problem that

he wants solved in terms familiar to him. A program called a COMPILER translates this representation into instructions that the machine can execute. One instruction in these new languages might be translated into many machine instruction. The programmer could now generate his programs much more efficiently, no longer having to deal with the actual details of the computer. Also, since the size (in source language statements) of each program was usually greatly reduced, the number of logical errors made by a programmer would be similarly reduced.

The other significant development was that of OPERATING SYSTEMS or PROGRAMMING SYSTEMS. Their purpose was two-fold. Firstly, operating systems provided subroutines to perform repetitive or intricate operations for the programmer. Many of these subprograms did routine mathematical calculations (such as square roots, exponentials, etc.). Often they related to input/output (I/O) operations. The programmer could now simply tell the operating system to read a block of data from a tape. The system would issue all the necessary commands to perform this and also initiate any error recovery which might be needed. The availability of all these routines freed the programmer from much of the tedious coding, leaving him free to do more creative work. Also worth noting is that since these routines were supplied, the user no longer had the

opportunity to make errors which might have occurred if he had to write the routines himself.

The second facility which operating systems provided on medium to large computers was the ability to BATCH jobs. A whole set of jobs could be made available to the computer at one time. When one job was complete, the operating system would start processing the next one. All information needed to process the job was supplied on the punched cards included in the job. This included such information as the type of language processor (Fortran or Cobol, etc.) and the external facilities needed (tape drives, etc.). Eliminating operator intervention between jobs cut out much machine idle time. The next step was to allow the operating system to actually run more than one job concurrently. This was done by having several jobs reside in memory at the same time. If one job could not execute any more instructions until an I/O operation was complete, the operating system would let one of the other jobs run for a while. This technique of running more than one job at a time is called MULTIPROGRAMMING.

These advances all helped to make more efficient use of the computer resources and, consequently, improved service and reduced the cost to the computer user.

The newer higher speed computers with their advanced software had many advantages over older models. Due to their speed, these machines could be used to tackle problems, which previously simply could not be handled. The new programming languages also helped as the effort needed to create a large program was often much reduced. The "cost/performance" ratio, the cost to perform a given amount of computing, tended to go down as the size of the machine increased. Of course since the number of applications using the computer increased, the overall cost often increased as well.

For many applications, the type of operating system just described was and is quite suitable. However, for some applications, specifically program development and execution of short jobs, and information retrieval (inquiry) operations, this environment was far from perfect. In many computer installations, a job requires a minimum length of time from the moment it is submitted for running until the output is available to the programmer. This time is seldom less than two hours and often as long as a day. This is true even if the actual job is very short. A programmer working on a project often needs many short runs to complete his work. The batch operating system cannot usually satisfy this type of need while at the same time provide the very sophisticated facilities needed by other users.

The dilemma of providing short turn-around for simple jobs was answered by the development of TIME-SHARING or TELEPROCESSING SYSTEMS. These systems usually work on the following principle. Instead of giving each programmer a ten minute time slot each day as a batch system might, this system might give him a one second slice of time every minute (on the average). To accomplish this feat, a number of typewriter-like terminals are connected to the computer. The operating system apportions the computer's time among all the active users at terminals. The user, instead of punching his programs on cards and submitting them to be run, now types the programs directly into the computer via his terminal. The operating system keeps track of what each user is doing. Since the average terminal user does not make very large demands on a powerful computer, the operating system, if properly designed, can keep a number of users satisfied simultaneously. Ideally, each user's requests or jobs can be completed quickly and while he is thinking about what to do next, many other user's requests are processed.

It is not inconceivable to imagine an operating system which could provide a wide range of computing facilities to the batch users, and at the same time, provide a comprehensive, efficient means of programming via a remote terminal. However, despite several valiant attempts, this

had not been accomplished (at the time of the study to be discussed) for an IBM S/360 computer at a reasonable cost to the installation and user.

CHAPTER 2

ILLINOIS BELL TELEPHONE

The developments in computer technology coupled with a changing set of data processing requirements necessitated a re-evaluation of the computer facilities at the Illinois Bell Telephone Company during 1969. At that time the processing at the company headquarters was being done on several relatively small Honeywell computers, an IBM 7074 and two IBM System/360's, a Model 30 and 40. The S/360-40 was running an interactive teleprocessing system (RAX)† during the day and doing batch processing in non-prime time using the IBM DOS (Disk Operating System). The S/360-30 was also running DOS. DOS is an operating system providing reasonably good facilities for business oriented applications on small to medium sized machines (the Model 30 and 40 fit this category). RAX is an independent system providing basic language and data management facilities for use from a number of remote terminals concurrently. Its facilities were (at that time) satisfactory for many applications particularly in the fields of engineering and data collection. At Illinois Bell, the largest application was data collection from remote locations. There was also a

---

†RAX - IBM program 360A-CX-17X. It was originally called "RACS" for Remote Access Computing System.

large amount of general interactive program execution and program development. Both the RAX and DOS IBM S/360's were quite heavily loaded.

There were several criteria and aims concerning modifications to and expansion of the computing facility. It was desirable to provide a better time-sharing service than the standard IBM RAX system. Also, it was necessary to be able to handle a much larger workload on this system. The intended plan was to replace the DOS system with the S/360 Operating System (OS). This would provide a much more versatile system featuring virtually all the batch oriented processing facilities which could be needed. OS is better suited than DOS to take advantage of a large computer configuration and to provide the users with a sophisticated set of facilities. This was needed since it was planned to eventually shift much if not all of the work of the Honeywells and the IBM 7074 to the S/360 computer(s). It should be remarked in passing that OS, because of its flexibility and wide range of services offered, has a much larger overhead than DOS, and thus generally requires a more powerful machine.

It seemed necessary to run both systems (or similar ones) since, at the time, no single system existed for an IBM S/360 which would provide the high level services and

through-put of OS and the teleprocessing facilities of RAX. Even if such a system had existed for another manufacturer's machines, the conversion effort would have been enormous.

The criterion of a better time-sharing system was met by acquiring the rights to utilize the McGILL-RAX operating system. This system is a version of the standard IBM RAX system[2], highly modified by the McGill University Computing Centre to provide a much improved set of language and terminal facilities for users.†

The requirements now evolved into the following. McGILL-RAX (or an equivalent system) must be run during the daytime hours (7AM-5PM). Its machine must be powerful enough to support a much larger workload than the current Model 40. The OS system must have the capability of processing a large amount of work relative to the current DOS - Model 40 setup. It must be possible to process at least test jobs and certain production jobs on OS during the day. This would allow programmers who are developing and testing programs to use their time more productively. Both systems should have some surplus power at the time of installation and must be expandable without any unreasonably high jump in cost.

†Throughout the rest of this paper, the name RAX will be used to refer to the McGill-RAX operating system.

Several proposals were made. They are outlined below.

Configuration one consisted of two S/360 Model 50's. During the day, RAX would be run on one and OS on the other. At night, both machines could be used for OS. One great advantage of this setup would be due to the fact that the CPU and much of the I/O equipment would be duplicated. The online RAX system would thus have excellent backup in case of the failure of some piece of the primary RAX machine. For several reasons however, this was not a very satisfactory configuration. Running two individual OS systems at night would be inconvenient since it implies maintaining two versions of the operating system, two sets of libraries, and so forth. A prime disadvantage of this configuration would be the very high cost of upgrading one or both of the Model 50s to a S/360 Model 65 (the next larger size CPU). The two Model 50s would be able to cope easily with the immediate workload, but should a larger machine be needed for either system, the cost increase would be large indeed.

Proposal two called for one S/360 Model 65. Such a system would run RAX during the day and OS for the remaining time. This hardware setup would give both RAX and OS plenty of room for expansion and also happens to be the least

expensive of the suggestions. There were two major drawbacks. A large amount of hardware, necessary to run OS, would be idle while RAX was running. RAX simply had no use at that time for all the core storage and I/O devices needed by OS. Also, and more important, no OS jobs could be processed during the day. This could not be accepted.

Solution three was to use one large S/360 Model 65 running OS. Although not utilizing RAX, this proposal quite nearly met all the requirements. Under the Operating System, a program called CALL/360 could be used. This is a system similar to RAX, the major difference being that it is not a stand-alone operating system but is part of OS. It is not quite as versatile as RAX, but it would fit the requirements at Illinois Bell. The major deterrents to this implementation were the higher hardware cost and the large amount of conversion effort (and thus cost) needed to modify all the RAX jobs to run under CALL/360.

The fourth solution was to use a Model 65 using the "Hypervisor" shared storage feature. This hardware enhancement, when coupled with the appropriate software, could allow two independent operating systems to coexist in the same machine, each operating system running as if it had complete control over its own computer. The Hypervisor software basically controls the two supervisors, thus the

name HYPERvisor. If RAX and OS could be Hypervised efficiently, this proposal would be quite attractive. At this point the McGill University Computing Centre was contacted concerning the feasibility of such a program. After the matter had been studied, it was decided that such a Hypervisor could be written. The Hypervised Model 65 could run RAX and a small OS system during the day and one large OS system at night. In this configuration, OS jobs could be processed during the day. Also, the large 65 configuration at night could, if properly used, be more powerful that two Model 50 CPUs. There were disadvantages of the Hypervised 65. Because of the hardware/software implementation, duplication of some hardware for RAX and OS would be necessary. When both systems were running, each must have control of its own channels and I/O devices. The possibility of a failure in one system impacting the other system also had to be considered.

An overall consideration was the fact that any Model 50 CPUs acquired would be rented while a Model 65 could be purchased. The economics are such that in the long run, it would be less expensive to purchase than lease, but it would not be worthwhile to purchase a machine which might soon be outgrown (i.e. a Model 50). Because of this, the 65 configurations were even more attractive than the rental prices indicate.

One other proposal was considered. This involved the use of a System/360 Model 67 CPU running an operating system called CP/67. A Model 67 is a special variety of S/360 with full relocation hardware.† The CP/67 system allows several conventional S/360 operating systems to run simultaneously. The system is very generalized. Because of this, it requires a large amount of hardware to support it. Also, the software was extremely inefficient (at least at the time this decision had to be made). This system would be able to run RAX and OS simultaneously but the overhead would tend to decrease performance by too much, especially considering the high cost of the hardware involved.

Table I summarizes the various configurations with respect to cost and features.

Based on the preceding data, all solutions but the OS-CALL/360 and the OS-RAX Hypervisor were ruled out. The final decision was to use the Hypervised Model 65 concept. The decision was made partly because of the conversion effort which would be necessary to use CALL/360. A large factor in the decision, however, was that Illinois Bell was

†Full relocation hardware allows any section of physical memory to be assigned any reasonable address (in 2048 byte blocks) by the the operating system supervisor. This allows the programmer to think he has contiguous storage, and permits the system to fully utilize fragmented storage.[3]

TABLE I

ILLINOIS BELL TELEPHONE

COMPUTER PROPOSAL SUMMARY

RAX	OS	COST	ADVANTAGES	DISADVANTAGES
50	50	$74,182	Backup hardware	Upgrade cost high
50	65	$92,652		Upgrade cost from two 50's
65	50	$88,682		Not a likely configuration
65		$66,013	Inexpensive more power than two 50's	No OS during day
Hyp 65		$73,642	OS during day	Two systems on CPU affects reliability
67		$81,903	Versatile	Expensive, slow extra versatility not needed.
65 CALL/OS		$77,148	One standard machine/ software	Conversion effort, etc.

extremely pleased with the RAX system which they had obtained from McGill. It was felt that if a Hypervisor were to be obtained from McGill, it would be of similar quality. The Hypervisor would effectively give a full S/360 Model 65 to RAX during the day, a large Model 65 to OS at night, and a "slow" Model 65 to OS during the day. Since both systems would have a rather powerful machine available, there should be no need to upgrade CPUs in the near future. The McGill Computing Centre agreed to write the software needed to Hypervise RAX and OS on the 65 (under contract to the Illinois Bell Telephone Company).

CHAPTER 3

SYSTEM/360 COMPUTER ARCHITECTURE

To understand the Hypervisor implementation, it is first necessary to have a good background in the System/360 internal architecture and operation.[4]

The main components of a S/360 computer are a main storage (memory), a central processing unit (CPU), channels and various input/output (I/O) devices.

The MEMORY of a computer is a device capable of storing binary information. The unit of storage in a S/360 is eight bits and is called a BYTE. Bytes in the memory are consecutively numbered starting at zero. Each number is considered to be the address of the corresponding byte. The memory is used for storing programs consisting of instructions to the computer, and data related to the problem being solved. On a S/360, the storage device is composed of magnetic ferrite cores. The term CORE has become synonymous with the term memory and will be used interchangeably.

The CPU is responsible for executing instructions stored in the memory. It can also establish communication

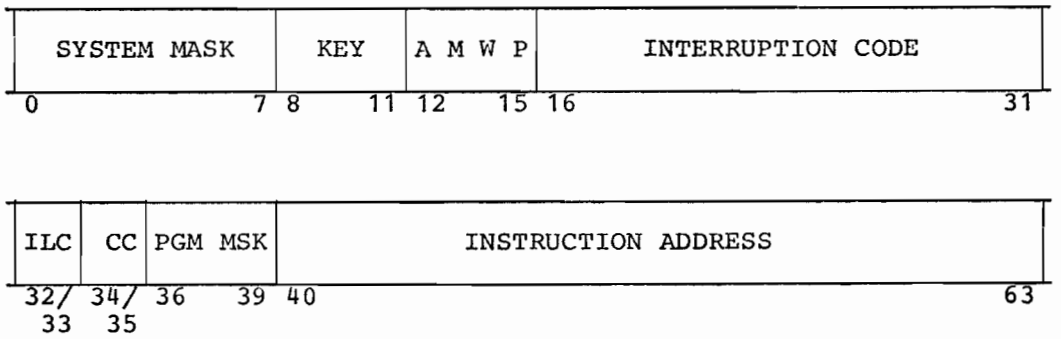
with the I/O devices under control of the program being executed. Associated with the CPU is the Program Status Word (PSW). The PSW contains eight bytes of information completely describing the status of the CPU in relation to the running program, at any given time (Fig. 1). It contains, among other things, bits of data specifying whether the CPU is running or waiting (in WAIT state), what type of occurrences may interrupt the current instruction sequence, and the address of the next instruction to be executed.

Under normal conditions, the CPU performs the operation requested by each instruction. When this is complete, it goes on to the next sequential instruction. This process continues until the instruction executed is a branch. In this case, the next and successive operations are fetched starting from the address pointed to by the branch instruction. The normal instruction sequence can also be disrupted by interrupts, as described in the following sections.

The channels interface the I/O devices to the CPU. When a program wishes to do some input or output, the CPU sends data to the channel concerning the type of operation to be done (read, write, etc) and the adresses in memory where the data is located or is to be placed. The channel

FIGURE 1

PROGRAM STATUS WORD



SYSTEM MASK
- Bits 0-6 If on, allows I/O interrupts on channels 0-6 respectively
- Bit 7 If on, allows external interrupts

KEY

Specifies memory fetch and store limitations. If all zeros, current program can fetch or store anywhere in memory

AMWP
- A not used
- M If on allows machine error interruptions
- W If on machine is not running, but in "WAIT" state
- P If on machine is in "PROBLEM" state and may not perform any supervisor functions.

INTERRUPTION CODE

After an interruption, the reason for the interrupt is stored here

ILC

Instruction Length Code - length of the last instruction executed (in a S/360, instructions can be 2,4, or 6 bytes long)

CC

Condition Code - two bits which may be set by one instruction and tested by later one

PGM MSK

Program mask - four bits related to interruptions which can occur during arithmetic operations

INSTRUCTION ADDRESS

Address of next instruction to be executed (this and the ILC can be used to compute the address of the last executed instruction.

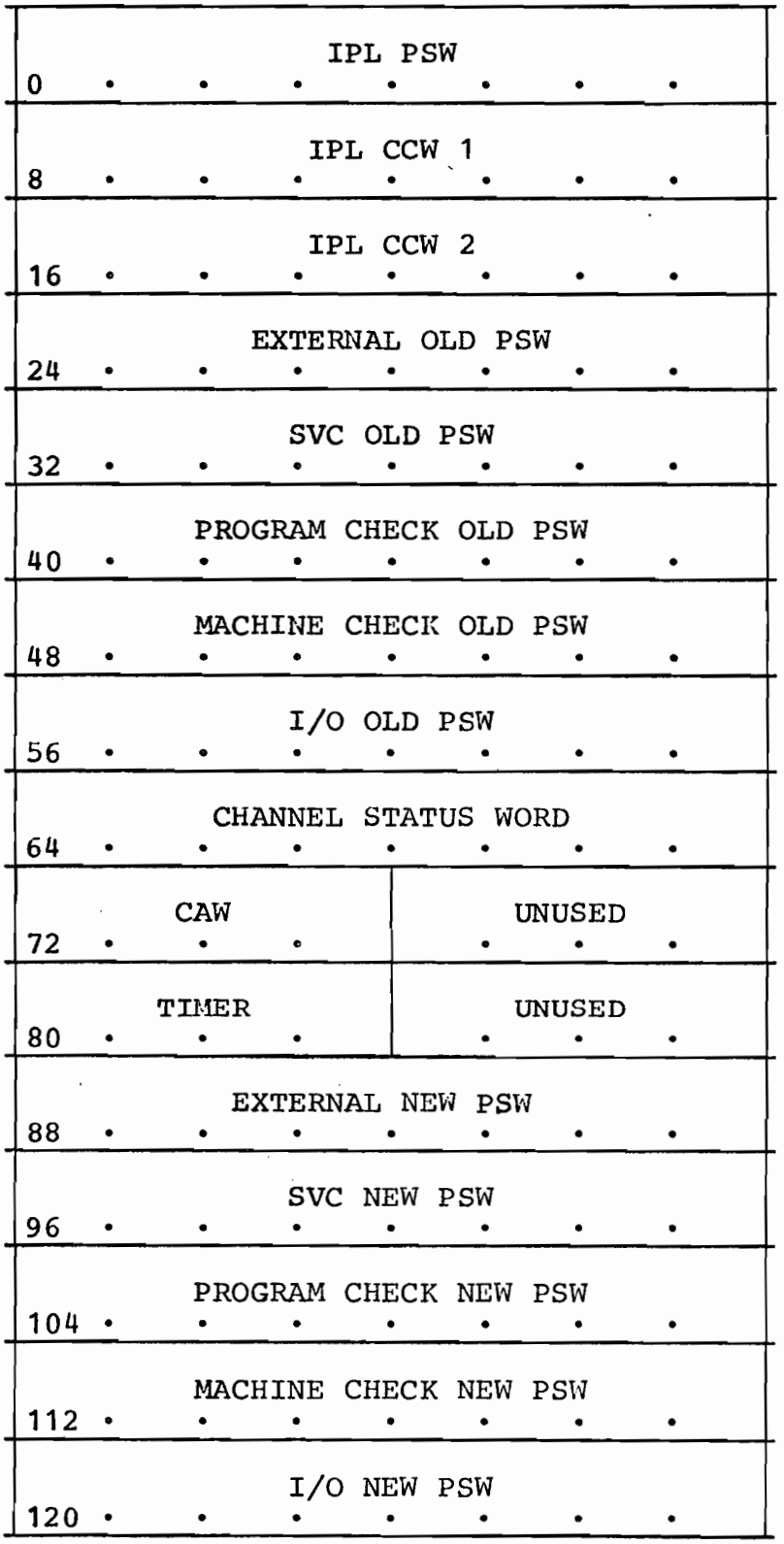
then operates independently and the CPU is free to continue other processing. When the I/O operation is complete, the channel signals the CPU that it is finished. The mechanism used to signal the CPU of a completed channel operation is the I/O interrupt. When an operation is complete, and the current PSW allows this channel to cause an interruption (Fig. 1 - System Mask), the current PSW is stored in a special location (Fig. 2) in the memory (I/O OLD PSW - loc 56).† Next, a new PSW is loaded from another specified location (I/O NEW PSW - loc 120). This new PSW contains an instruction address pointing to a section of program designed to handle completed input/output operations. This routine can take whatever action is necessary and then resume what the CPU was doing before the interrupt by loading the PSW that was stored at location 120. The channel address is a number (usually from 0-7) by which the channel is identified. If when an I/O operation completes, the current system mask disallows this channel's interrupts, the interrupt remains "PENDING" until the system mask is changed to allow it. The interrupt then occurs just as described.

The principles employed in performing an input/output operation are quite straight forward. The main

†All memory locations referenced in this paper are expressed in decimal.

FIGURE 2

SYSTEM/360 LOW CORE MAP



means of communication from the CPU to the channel is the Channel Command Word (CCW). This consists of eight bytes of information including the type of operation to be done, the storage address of the data involved, and the number of bytes of data to be transferred. To initiate an I/O operation, the software constructs one or more CCWs and places the address of the first one in the Channel Address Word (CAW) at loc 72 of the memory. The program then issues a Start I/O (SIO) instruction specifying the unit number of the device to be used. The channel looks at the pointer in the CAW and from this location (i.e. the location of the CCW) gets its instructions. Assuming that the CAW and CCW are valid, the SIO instruction is now complete and the CPU is free to go on to the next instruction. The channel performs the operation(s) described in the CCW(s).

As soon as it is finished, and as soon as the CPU System Mask allows interrupts from this channel, an I/O interrupt occurs. The unit number of the device which was addressed is stored in the interrupt code of the I/O OLD PSW. Also, a Channel Status Word (CSW) is stored at location 64. The CSW indicates whether or not the operation was successful. If it was not, the CSW also gives information pertaining to the error conditions.

The interval timer is another feature available on most S/360 computers. This device allows the program to measure and keep track of time. The interval timer consists of four bytes of storage starting at location 80 in the memory. Every 1/60 of a second, this location is decremented. The amount of each decrement is 1380. This is equivalent to 1/60 second, since the interval timer is in units of about 13 microseconds. If this value should become negative, a timer interrupt occurs (assuming the current PSW system mask allows it).

This timer interrupt is a specific type of EXTERNAL interruption. It operates in a way similar to the I/O interrupt; in this case however, the machine stores the current PSW in the EXTERNAL OLD PSW (loc 24) and fetches the new one from the EXTERNAL NEW PSW (loc 88). This new PSW points to a routine written to handle timer interrupt conditions.

To use the timer to measure elapsed time, the program need only initialize the timer to some value. At any later time, the difference between this original value and the current contents of the timer is a measure of how much time has passed.

To limit a function to a specific length of time, a program places in the timer location a value equal to the allowed time (in timer units). When the period has expired, a timer interrupt will occur.

The last feature of the S/360 machines which is relevant to the Hypervisor design is the method in which a program or operating system is first loaded into the computer. To begin operation, the address of the device where the program resides is placed in a set of dials. The "LOAD" button is then pressed. This causes the INITIAL PROGRAM LOAD (IPL) procedure to take place within the machine. This procedure is equivalent to doing a SIO on the selected device, using a CCW which contains a read command. This reads 24 bytes of information into locations starting at location zero of memory (Fig. 2). This 24 bytes of data from disk contain more CCWs and a PSW (Fig. 2). The channel program continues after the read, using the CCWs now at location zero. When the I/O operation has been successfully completed, the PSW at location 16 is loaded. This PSW normally points to the program loaded by the I/O operation just performed.

The Hypervisor hardware modification allows the memory of the computer to be divided into two independent sections from the viewpoint of the programs. In the

following explanations and examples, for simplicity, we will assume that the computer has a memory of 20000 bytes (in reality, the core sizes of machines on which this paper is centred are from 256K to 1024K (K = 1024 bytes)). When it is divided, there will be two segments, each of 10000 bytes. Note that the memory addressing starts at zero, so that the 20000 byte memory is addressed from 0 to 19999 and the 10000 byte segments as 0-9999. When the Hypervisor feature is active, the memory is partitioned into two logically separate core storage boxes. As far as the CPU and channels are concerned, the addressing of each box starts at location zero and increments by one until its highest address. Thus there are now two locations "0", two locations "1", etc. Since there are now two low core areas, all special locations (e.g. OLD, NEW PSW's) are duplicated. It is the joint function of the Hypervisor hardware and software to control which of the core boxes is to be used for which purposes.

This control is achieved via the PREFIX CONTROL REGISTER (PCR). The PCR is an extra hardware register consisting of eight bits numbered 0-7. Bits 0-6 refer to I/O channels 0-6. Bit 7 refers to the CPU itself. If all the bits are zeros, the machine operates as a normal S/360. No memory partitioning take place. Once any of the bits are on, core storage is divided into two segments. If the bit

corresponding to a channel is on, then any time that channel references the computer memory, 10000 will be added to the address specified by the channel. What this means is that if the channel is instructed to read some data into location 100, and its PCR bit is on, the data will really be placed at location 10100 of the real memory. Note also that when this channel stores its CSW and fetches it's CAW, although it is referring to locations 64 and 72 respectively, the real memory locations used will be 10064 and 10072. As far as this channel is concerned, its memory starts at byte zero and goes as far as 9999. In reality, it is using locations 10000 to 19999.

Similarly if bit 7 of the PCR is on, any storage reference made by the CPU is relocated. This is true for storage references to get instructions as well as data. A branch to location 500 will cause the next instruction to be fetched from real core location 10500 if PCR bit 7 is a one.

If the bit for any channel or the CPU is off, it can refer to memory locations 0-9999. If the bit is on, apparent addresses are also from 0 to 9999, but these latter references will actually address physical locations 10000-19999. It can be seen that a program written to work in a normal machine can work in the upper core box of a Hypervised machine without any modification. All that is

necessary is to make sure that the relevant PCR bits are set to ones.

Table II gives examples which illustrate these rules. Instead of the 20000 byte machine used in the previous examples, the table assumes a machine of 50000 bytes partitioned at byte 20000.

Note that the bit 7 for the CPU can be turned on or off at will. However, if the channel bits are changed while data is being transferred, the results could be disastrous. Data would start in one of the memory boxes and suddenly jump into the other one. The channel bits could be changed if all channel activity could be quiesced first. However, this would require much co-ordination and synchronization between the two programming systems. In most cases, it is simply not practical. It also follows that since the channel bits should not be set to zero during normal multisystem operation, once the memory is partitioned, it must remain so. This means that while running, a program residing in one part of the memory cannot access the core of the other.

Remember also that the interval timer is at location 80 of the memory and is decremented every 1/60 of a second by the CPU. If the CPU PCR bit (bit 7) is on at the

TABLE II

HYPERVISOR SHARED STORAGE FEATURE

PREFIX CONTROL REGISTER EXAMPLES

PCR	REFERENCE FROM	RELOCATED	ADDRESS	ACTUAL ADDRESS
10100001	Channel 0	YES	100	20100
10100001	Channel 1	NO	100	100
10100001	Channel 2	YES	200	20200
10100001	CPU	YES	300	20300
10100001	CPU	YES	20400	40400
10100001	CPU	YES	30400	invalid 50400>49999
10100001	Channel 0	YES	20100	40100
10100001	Channel 1	NO	20100	invalid 20100>19999
11000000	CPU	NO	201	201
11000000	CPU	NO	30003	invalid 30003>19999
00000000	Channel 0	NO	100	100
00000000	Channel 1	NO	30004	30004
00000000	CPU	NO	45777	45777

time, the location that gets decremented is the location 80 in the high core box. The machine is now running with two timers. One of them being decremented part of the time and the other active the remainder of the time.

One new instruction is added to the System/360 with the Hypervisor feature. It is the SET PREFIX and BRANCH (SPB) instruction. The operands of the SPB instruction are one 8 bit byte and a memory address. The PCR is first set according to the 8 bit code. Then the CPU branches to the address specified in the instruction. If the new PCR contains a zero in bit 7, the address is used as is. However, if bit 7 is a one, the address will refer to the upper core box since relocation is now in effect for all CPU storage references. This instruction therefore gives the ability to set the Prefix Control Register and to transfer program control from one of the core boxes to the other.

CHAPTER 4

McGILL-RAX - OS/360 HYPERVISOR DESIGN

The concept of Hypervising two operating systems was first developed to allow a S/360 computer to run both OS/360 and a 7074 Emulator program simultaneously. An emulator is a hardware assisted program which simulates another type of computer. The 7074 emulator runs on certain S/360 CPUs. It is a stand-alone system, that is, it normally runs on a machine all by itself, not depending on any other operating system. When a computer installation converts from an older type of computer such as an IBM 7074 to a newer S/360, the emulator allows old jobs to be run without any modification or conversion. As time passes, less of the workload is emulated and more becomes S/360 programming. To run this type of job mix the computer must alternatively run OS and the emulator program. The Hypervisor hardware and software allowed these systems to be run concurrently.

Since the initial implementation, the Hypervisor has been used to run several other combinations of operating systems. Examples of these are 7074 emulator/DOS, two 7074 emulators, and two DOS systems.[5] It has also been reported that a modified IBM RAX and OS/360 have been Hypervised

before, but documentation relating to this has not been available. A short paper[6] was obtained (from IBM) giving an implementation plan for Hypervising RAX and OS. Because of different design criteria, this paper's ideas were not followed very closely, however, it was useful in the initial planning stage. It was never established whether this paper described an existing system or was simply a possible implementation specification.

Once the decision was made to use the Hypervised S/360-65, the software specifications were set. There were two sets of guide lines established. The first set were criteria which it was felt must be met for the Hypervisor to work as planned. The other set were desirable attributes which were to be included if possible. That is, the first criteria must be met, the second were to be satisfied but not at the cost of any of the first set.

Major Criteria

A-1 The RAX system must have priority over the OS/360 system. Any time RAX needs the CPU, it must get control over it quickly (i.e. within a period of time measured in microseconds). There must be a minimum of degradation in RAX due to OS.

A-2 While running under the Hypervisor with OS, RAX reliability should not suffer appreciably. This means RAX should be able to keep running even if OS crashes (stops dead).

A-3 RAX time charges must be accurate both for execution and connect time. This means RAX's timer must be running (logically if not physically) at all times. RAX must be able to keep track of time used for CPU work, time in wait state, and time of day, all reasonably accurately.

A-4 OS must be able to maintain job timing. OS charges are by CPU time and a measure of how long the job utilizes system resources (core, I/O units, etc). Time of day accuracy is desirable but not mandatory.

A-5 The OS system will have only 256K memory during the day. This is not a very large configuration and therefore the amount of core occupied by the Hypervisor on the OS core box should be kept to a minimum.

Minor Criteria

B-1 The Hypervisor should depend as little as possible on either control blocks or coding within OS/360. A new version or release of OS should necessitate a minimum of changes in the Hypervisor.

B-2Similarly, the number of changes to OS/360 itself should be kept small, preferably zero.

B-3 The Hypervisor coding should be efficient, particularly the sections which are executed often (up to several million times per day). It should not appreciably degrade either RAX or OS operation.

B-4 The same system residence disk packs for both OS and RAX should be able to be used on a Hypervised system or stand-alone. It would not be desirable to have to maintain two almost identical systems.

B-5The Hypervisor should keep internal usage statistics. These are useful to measure its performance and help indicate areas where design improvement might be needed.

B-6 It should be easy to modify the Hypervisor to reflect hardware configuration changes.

After the initial installation of the Hypervisor, a new requirement developed. In the original design, if OS failed catastrophically during the day, it was felt that it would be satisfactory to leave OS off until the end of the RAX day. This seemed reasonable at the time for several reasons. First, it was not expected that OS would fail very often. Second, since before the Hypervisor was used, no OS production was processed during the day, it was felt that to occasionally revert to this condition would not hurt greatly.

In practice, OS did fail with some regularity due to both hardware and software problems. Also, once the programmers were used to having OS jobs run during the day, it was not very satisfactory to suddenly withdraw this facility. The new criteria all related to increasing the reliability and availability of both systems, but particularly OS.

Additional Criteria

C-1 The Hypervisor must be able to re-IPL OS during production hours with minimal effect on RAX and its users.

C-2 Either system can run without the other. That is, RAX can run under the Hypervisor without OS/360 running, and OS can run if RAX is inactive.

C-3 Either system can be stopped or started at any time.

These additions would not only allow either system to be re-IPL'd during the day, but due to the second and third specifications, if one of the systems could not be started due to either hardware or software malfunctions, the other system could still be run. When the errors had been corrected, it would be very likely that the first system could now be IPL'd without affecting the work being processed on the running system.

CHAPTER 5

HYPERVISOR IMPLEMENTATION

The Hypervisor program to be described was written in accordance with the specifications laid out in the previous section. It corresponds to the McGILL-RAX - OS/360 Hypervisor currently installed at the Illinois Bell Telephone Company.

External Implementation

From the machine operator's point of view, the Hypervisor operation consists of the following procedures. To start, the Hypervisor is loaded from a magnetic tape or a deck of cards. Once it is in core, a command can be typed in on one of the console typewriters. This command should instruct the Hypervisor to IPL either OS or RAX. The specified system will be loaded. When its initialization is complete, the console of that system can be used to instruct the Hypervisor to load the other operating system. After that, commands can be entered on either of the systems consoles.

Theory of Operation

When both the systems are running, RAX has ultimate control of the machine. When RAX has no computing to do, and it would normally go into wait state, it now transfers control to the Hypervisor. The Hypervisor restarts OS/360 at the place where it was last interrupted. When an I/O interrupt occurs, it is handled by a Hypervisor routine. If it is for OS, a branch is taken to the OS I/O First Level Interrupt Handler (I/O FLIH). If it is from a RAX device, the information pertaining to the interrupt is passed back to the RAX machine for processing. That in principle is all the Hypervisor does. In practice, there are many complications.

The aim of the implementation is to give OS as much time as possible without degrading RAX. That is, any time during which RAX would normally go into wait state, let OS use the CPU. However, often when RAX wishes to go into wait state, its PSW system mask does not allow interrupts on all channels. If it were in real wait state and an I/O operation on one of the masked out channels completed, the interrupt would remain pending until RAX changed its system mask to allow this channel to interrupt the CPU. That is, the interrupt would not occur until RAX wanted it, and then it would occur immediately. With the Hypervisor, when

control is passed to OS, this interrupt will occur immediately, as OS allow interrupts on all channels. To stop OS from accepting interrupts on some channels some of the time would have necessitated extensive (if not impossible) modifications to OS. This was completely ruled out. Thus this interrupt will occur and be intercepted by the Hypervisor I/O interrupt routine. It cannot be passed on to RAX, since RAX does not want it now. This problem is solved by means of the Hypervisor being able to remember interrupt information until RAX is able to accept it. These interrupts are said to be queued or STACKED. The stacks are really First-In-First-Out (FIFO) queues†, one being maintained for each channel.

The other major bottleneck in the Hypervisor implementation has to do with the interval timer management. RAX is required to be able to keep track of time accurately. The timer at location 80 of the RAX memory, however, is only running when RAX is actually using the CPU. Somehow, RAX must keep track of the rest of the time when OS is in control. Basically this is done by noting OS/360's time of day (TOD) whenever OS is given the CPU, and again before RAX is to get it. The difference is subtracted from the

†The term STACK normally refers to a first-in-last-out list, but it is used in this paper to mean a FIFO list to coincide with the terminology used within the actual Hypervisor module.

original value at RAX's location 80. This effectively lets RAX know about all the time that passes while it is not running. Of course, it is not that simple. If the subtraction cause the interval timer to go from a positive value to a negative one, an external (timer) interrupt should occur. This is what would happen on the real hardware if the timer became negative. Similarly, if the Hypervisor finds that the value of the timer becomes negative, it simulates an external interrupt just as it normally sends I/O interrupts to RAX. If RAX is not currently allowing timer interrupts to occur, it is stacked on a special queue.

Another problem which occurs is that on a real machine, if an interrupt is pending but not allowed due to the PSW system mask, and later the system mask is changed to allow interrupts on this channel, the I/O or external interrupt is automatically taken. However in our case, this interrupt may have been accepted while OS was running and queued in the Hypervisor's stacks. When control is given to RAX, the information remains queued. Later, RAX may execute a SET SYSTEM MASK (SSM) or LOAD PROGRAM STATUS WORD (LPSW) instruction to change the system mask. The physical channel at this time would pass the interrupt on to RAX. Since it is pending in the Hypervisor stacks now, not the channel, RAX is not informed of its presence automatically. The Hypervisor has to tell RAX about it. The solution taken, is

to modify RAX so that any time it executes a critical SSM or LPSW instruction, it calls a subroutine in the Hypervisor program to inquire whether any pending interrupts can now be taken. If any are available, they will be presented to RAX at this time.

The next set of problems centre around the IPL of OS/360 while RAX is running. The actual IPL is simulated rather simply and will be described in the detailed program description. The main point worth noting concerns the timer management. When control is first given to the OS IPL program, the OS timer routines are not yet loaded into core yet. This makes it difficult to calculate the OS TOD. This time is needed to maintain the RAX clock. Fortunately, it is known that OS will very quickly set its time of day to 00:00:00 (i.e. zero hours, zero minutes, zero seconds). This gets over the first hurdle of the actual IPL. Later, (perhaps several seconds or minutes), the computer operator issues a SET CLOCK command to OS to tell it the real clock time. When the Hypervisor next gets control, it finds that the OS TOD has jumped by a large amount (from 00:00:00 to some large value). In reality, only a fraction of a second has gone by. The Hypervisor cannot show this jump to RAX.

Yet, it must be capable of passing large increments to RAX at other times.† The solution is that the Hypervisor ignores a large jump (greater than 30 minutes) only if it occurs when the previous time was small (less than zero hours, ten minutes). This method is not fool-proof. If OS is IPL'd under the Hypervisor between midnight and 12:30 AM, it will not work properly. However since the Hypervisor is normally working only during the day shift, the method outlined above will work quite satisfactorily.

†This would be necessary if RAX was on the air, but not very busy for a long period of time.

CHAPTER 6

HYPERVISOR INTERNAL DESIGN AND OPERATION

Following is a detailed description of the Hypervisor control program. Emphasis will be placed on logic flow within the software rather than coding techniques. The text closely follows the Hypervisor flowcharts found in Appendix A. Capitalized names within the text usually refer to labels used within the flowcharts.

Hypervisor Initialization

After the Hypervisor is loaded, it takes complete control of the machine by setting all the new PSWs in OS's low core to point to its own interrupt handlers. At the same time, the location 80 timer is initialized. A read command is started on the console typewriter. When the operator has finished typing something in, control is given to the standard Hypervisor console command handler. If this message is a request to IPL either RAX or OS/360, it is carried out. If any other reply is made, the initialization process must be re-done (i.e. IPL the Hypervisor again).

I/O Interrupt Supervisor

When an I/O interrupt occurs while OS is running the Hypervisor IOINT routine is entered. If the interrupt is from an OS/360 device other than the console typewriter, the OS I/O FLIH is given control. For the typewriter, tests are made to determine if the interrupt concerns a Hypervisor message. It could be either a Hypervisor command from the operator or an indication of the completion of a Hypervisor message to the operator. If it is, the correct processing routine is called. Otherwise, a transfer is made to the OS interrupt handler.

If the interrupt is from a RAX device, the information must be passed on to RAX or stored for future use. First though, the amount of time since RAX last had the CPU is calculated and subtracted from the RAX interval timer value. If this causes it to become negative, and external interrupts are allowed by RAX, the I/O interrupt is stacked and control is returned to RAX signalling a timer overflow. If it goes negative but external interrupts are not wanted, it is stacked. In all cases except when the timer interrupt is taken, a decision must be made concerning the original I/O operation (and any other ones which may already be stacked). The queues are inspected and if an

interrupt can be passed on to RAX, it is done. If none can be found, control of the CPU is returned to OS/360.

Interrupt Queue Management Routines

All pending interrupt data is stored in queues (or stacks). A queue exists for every channel used by RAX and for external interrupts. If a queue is non-empty, it consists of pointers to queue elements which contain the relevant information. There is also a dummy queue consisting of unused (free) queue elements. All the queues are managed on a first-in-first-out (FIFO) basis. Each queue control block contains data relating to the start and end of its stack. Also maintained is the count of elements currently on the queue and several usage statistics.† Each queue element has room for the device address where the interrupt originated, the CSW, and a pointer to the next element on the queue.

A set of subroutines is used to maintain the queues. The STACKIO routine will transfer information relating to the current I/O interrupt to the appropriate channel queue. During this operation, a call is made to a

†Statistics maintained are; the number of times the queue is used, and the all-time minimum and maximum number of elements in the queue.

subroutine within the RAX supervisor to fetch the contents of the RAX CSW. Since this was a RAX I/O interrupt, the channel PCR bit is on and the CSW is stored in the upper core box. The Hypervisor, which resides in the lower core box cannot access the CSW itself.

The STACKEXT routine notes that a timer interrupt has occurred. This routine will only allow one element in the stack. Two pending external interrupts is a condition which cannot occur on a real machine and should never occur within the Hypervisor.

The UNSTACK module inspects the pending interrupt stacks and returns with the best one which can be taken. Consideration is given to the current RAX PSW system mask, the relative priority of the channels and the order in which the original interrupts occurred.†

There is also a small function to simply empty out all the interrupt stacks (CLEANSTK). It is equivalent to a "SYSTEM RESET" on a CPU as it removes all traces of pending interrupts. It is used by the IPL simulator to obtain a "CLEAN" machine.

---

†External interrupts have the highest priority. Channels come next, the higher the channel number, the lower the priority. Elements within each queue are processed FIFO.

The queue utility subroutines GETQEL, PUTQEL are used by all the above functions to perform the actual queue element additions and deletions.

RAX Interface Routine

This section of the Hypervisor is entered from RAX to perform a variety of services. All communication from RAX to the Hypervisor is done via this path. There are three reasons for coming here.

The first is a request to inspect the stacks for pending interrupts which can now be taken. It is called after RAX executes a SSM or LPSW instruction which may change the system mask. Return is made to RAX with the information requested.

The second entry is used to pass to the Hypervisor a message from the RAX console. Control is given to the console command processor.

The third type of entry is used when RAX has nothing to do and therefore wishes to go into wait state. If any stacked interrupts can be returned immediately, they are. If not, the machine will normally be given to OS. First however, a check is made to see that the interval

timer has not gone negative since external interrupts were last allowed by RAX. If it had and control were given to OS, OS would at once receive a timer interrupt which rightfully belongs to RAX. The check is made by momentarily allowing external interrupts. If one occurs, it is either stacked or passed on to RAX (if RAX is currently allowing externals). If this is the first time the wait routine has been called, the opportunity is taken to send an IPL COMPLETE message back to the OS console, assuming it was OS that IPL'd RAX. In all other cases, OS is given the CPU. It keeps it until the next I/O interrupt occurs at which time IOINT takes over.

OS/360 - Hypervisor Interfaces

There are three Hypervisor routines that have contact with OS/360 directly.

The first of these is the OSTIME module. Its function is to give the I/O interrupt handler the time of day according to OS/360. Is OS is active at the time, the OS timer control blocks are inspected to determine what OS thinks the time is. Note that this will normally not be the correct time since the OS clock is not running while RAX is active. If OS is not being used, a fake time of day is produced. This is just the negative of the interval timer

(divided by 256 to obtain 300th of a second). Both of these times are not true clock times although the former is slightly better than the latter. However, on two successive inspections, with OS using the CPU in between, they both give accurate elapsed time. This is their true purpose and both are quite satisfactory.

The next two routines are used only once each during normal operation. One is IOCATCH. Its purpose is to intercept the first I/O interrupt when OS starts running after an IPL and perform patches to the OS nucleus. When OS is loaded, the I/O NEW PSW points to its own I/O FLIH. The first instructions of the interrupt handler however are modified to branch to IOCATCH. This routine replaces the modified instructions in the I/O FLIH, saves the original I/O NEW PSW and stores a new one at location 120 to give control to IOINT on all following interruptions. Also modified is the PROGRAM CHECK NEW PSW so that if the Hypervisor accidentally gets a program error, it will be noted as such instead of being attributed to some user program running under OS.

The last OS interface module is OSWAIT. Its function is to intercept OS/360 wait states. That is, when OS intends to go into a wait state, it instead goes to OSWAIT. The first time it is entered after start-up, if the

IPL request had been from RAX, a message is sent back to the RAX console saying IPL COMPLETE. This effectively returns control of the CPU back to RAX, since the OS IPL was done only as an exit to the RAX console handler. Under any other conditions, OSWAIT simply goes into wait state by loading a PSW with the wait bit set to one. This PSW, like the normal OS wait state PSW has a system mask of all ones to allow all interrupts.

Hypervisor Message Handler

If an operator command for the Hypervisor is typed in, control is given to the appropriate analysis section.† Either RAXMSG or OSMSG is entered, depending on which console the message was typed. Through the RAX console, the operator can drain (stop dead) RAX, drain OS, and IPL OS. From the OS console, the operator can drain and IPL RAX, and ask for a display of the Hypervisor internal counts (statistics). Also, at initial Hypervisor IPL time, this console can be used to IPL OS. When a valid command is recognized, the message scanner transfers to the correct command processor. If an invalid command is entered, an error message is returned.

†All commands for the Hypervisor are prefixed by two equal signs, as in '==IPL OS'.

RAX IPL Simulator

The purpose of the IPL simulator is to load RAX into the upper section of the memory, set the PCR correctly and transfer to RAX. It basically simulates the "LOAD" button with additional functions necessitated by the shared storage feature. Since there is no RAX I/O active at this time, the simulator sets the PCR to zero so that all core may be addressable. After clearing RAX's core, it moves into it a small IPL program. The CLEANSTK routine is called to discard any pending interrupts. Switches are set to say that RAX is being started. The PCR is set for all RAX channels and the CPU. At the same time, a branch is taken to the IPL program in the RAX core box. This program starts a read request on the RAX system residence disk. After the read, the CCWs just read into core at location zero are executed. On successful completion, the PSW at location 16 is loaded. This completes the IPL of RAX. If the I/O had not been without error, the IPL program would go back to the Hypervisor and an error message would be sent.

Once RAX completes its normal system initialization, when it would normally go into wait state (waiting for work to do), it returns to the Hypervisor via the normal RAX wait processor. At this time, it passes to

the Hypervisor, addresses within the RAX supervisor of several service routines to be used by the Hypervisor.

OS/360 IPL Simulator

The program to load OS/360 is similar to the RAX loader but a bit more complicated. Since both the Hypervisor and OS reside in the lower core box(es), there is no need to set the PCR here. Channel programs are set up similar to those for loading RAX, and the I/O performed. If an error occurs, a message is issued. If the I/O is good, several patches are made to the OS IPL-TEXT program before it can be given the CPU. First, a byte is set to say it should load an alternate nucleus instead of the standard one. This secondary OS nucleus contains patches which cause branches to the OSWAIT and IOCATCH routines. Next a modification is made to limit OS to the memory below where the Hypervisor resides. If left on its own, OS would compute how much core was available. Since the space where the Hypervisor exists is physically available, OS would use it, thereby destroying the Hypervisor. The last set of patches is made so that if the IPL-TEXT program finds any serious errors (I/O or otherwise), where it would normally go into a dead wait state, it will now go back to the Hypervisor. Thus if RAX is active, it can keep on running. When all this has been completed, the IPL-TEXT will be

entered. OS will have complete control of the CPU until it first enters into its normal wait state. Then OSWAIT will be used to give CPU control back to the Hypervisor. The whole IPL procedure normally takes only a few seconds. It is not unusual for RAX to lose control of the machine for only about five seconds if the computer operators perform their jobs well.

Display Hypervisor Statistics

The HCOUNTS routine types on the OS console a series of lines each containing the name of an internal counter and its value. Both systems continue running while this is being done. OS however cannot access its console during this time. As it might take up to one minute for all the information to type, OS could stop due to this if it had some urgent message to type or needed an operator reply to continue. It is not a major problem since the counts are not normally produced very often.

Drain Processing

If the Hypervisor is instructed to shut down either operating system (via the "==DRAIN OS" or "==DRAIN RAX" commands), appropriate switches are set to ensure that the referenced system will not be made active again (unless it

is re-IPL'd). If OS has gone into a dead wait state or a loop, and the console typewriter cannot be used, the PSW RESTART button on the CPU control panel can be depressed. This causes a branch to KILLOS which has the same effect as entering a "==DRAIN OS" command.

Console Exit Processing

After processing any operator command, the Hypervisor exits back to the system which originated the message. If it was RAX, the RAX console processor is returned to; if OS, after any reply has been typed, the following subterfuge is carried out. Since the original command was intercepted by noticing that an OS/360 console read operation had completed, OS still thinks the read is active. To satisfy the OS console manager, an I/O interrupt is simulated. A "DISPLAY TIME" command is placed in the original buffer in place of the line the operator really typed. The I/O PSW and the CSW is set up to indicate the completed read operation. This simulated I/O interrupt is handled as if it were a real one. This method logically completes the read which OS/360 initiated and when OS responds to the message, it effectively time-stamps the Hypervisor command.

Hypervisor Storage Requirements

One of the major criteria in the Hypervisor design was that it occupy as small a region of main storage as possible. The final version of the Hypervisor described here requires only 5500 bytes of memory. Since the value must be rounded up to 2K (2048) segments, the final storage requirements are 6K. This was well within the original specifications.

CHAPTER 7

OPERATING SYSTEM MODIFICATIONS

OS/360 Modifications

The design criteria called for the Hypervisor to be as independent of OS/360 and its internal workings as possible. It was of course necessary to assume that the basic operations of a S/360 CPU would not change. The first version of the Hypervisor was very OS independent. To maintain TOD, a small routine accessed the relevant OS control block.[7] If the format of these would be changed in the future (quite unlikely), the Hypervisor subroutine could easily be rewritten. In general however, it was most unlikely that future versions of OS/360 would not work properly with the Hypervisor.

In the final Hypervisor, due to the need to IPL OS/360 and yet still retain control of the machine, much more knowledge of OS had to be built into the Hypervisor logic. Also, OS itself had to have several changes made (although none very large or complicated). There are two areas of problems.

1. When OS is IPL'd, it brings into core its own set of new PSWs. This cannot be stopped easily as this is only a minor result of the necessary loading process. Thus I/O interrupts no longer go to the Hypervisor but to the OS I/O FLIH. To retain control, the Hypervisor must inspect ALL I/O interrupts.

2. During initial loading, if some sort of error is found, the load routine simply loads a PSW to put itself into wait state with no interrupts allowed (Dead Wait). This lack of any activity acts as a signal to the computer operator that something is wrong. If this would happen under Hypervisor control, RAX would be dead also. This condition is not desirable.

The solution to both these problems is not difficult. In the first case, the initial instructions of the OS I/O FLIH are modified to transfer to the Hypervisor. This Hypervisor routine saves the OS new PSWs and replaces the with Hypervisor PSWs. At the same time it replaces the original instructions in the OS I/O FLIH. All I/O interrupts now go to the Hypervisor and then are passed on to the I/O FLIH in the appropriate system as described previously.

The second problem also has a relatively simple solution. After the Hypervisor reads into core the OS/360 initial load program, it modifies it to go to a Hypervisor error routine instead of stopping dead in the event of any serious error. At the same time, the IPL program is modified to make it seem that the amount of core available to OS/360 ranges from the bottom of memory up to but not including the Hypervisor program.

All of the actual changes to OS are made in an extra copy (secondary nucleus) and are only loaded into core when the Hypervisor IPLs OS. When OS is loaded normally by the computer operator, an unmodified OS is used.

RAX Modifications

Just as in the case of OS/360, some changes to the RAX system were necessary for the operation of the Hypervisor. It was desired to minimize these modifications to make conversion to any future version of RAX easy, but this was not as pressing a need as with OS.†

†The installation at Illinois Bell of a new McGill-RAX version would be infrequent, and even then, it would be done by people fully qualified to do the necessary Hypervisor modifications.

There were three major design features whose implementation required changes to RAX. In addition, there were several relatively minor facilities also added.

1. The first change in RAX is the one around which the whole system revolves. The RAX routine which normally loads a wait state PSW to put RAX into wait state (during idle moments), now transfers control to the wait routine in the Hypervisor. This coding gives the CPU back to OS/360 at the place where it was last interrupted.

2. A standard RAX system, during most of its running time, allows interrupts (via PSW system mask) on all channels. If an interrupt from an OS device were to occur, an ordinary RAX system, knowing that this device does not belong to it, would simply discard this interrupt. This is certainly not an acceptable method from OS's viewpoint. There are three possible alternatives to this. First RAX, on recognizing an interrupt from OS, could pass it on to OS, thus giving up the CPU. This would violate the rule that of RAX having ultimate priority. OS should only get control of the machine when RAX can no longer do any processing. The second solution is for RAX to queue this interrupt for OS just as the Hypervisor does for RAX interrupts (those which

cannot be accepted immediately). The complications of this method (particularly with regard to dequeueing these interrupts) ruled it out. The third method is quite simple and it is the one used. While RAX is running, it now, never sets a system mask to allow interrupts on the channels belonging to OS. Since OS's interrupts are never allowed during RAX operation, they bother no one. When OS gains control of the CPU next, it sets a system mask of all ones and any pending interrupts fall through.

3. The third problem and its solution is by far the most complicated. When running on a regular machine, any interrupt which attempts to occur while the machine is in wait state with the system mask disallowing this channel's signals (or external interrupts), will remain pending. Whenever the system mask is changed to allow this type of interrupt, it will occur immediately. Under the Hypervisor, RAX does not go into wait state but goes to OS/360. This interrupt just mentioned actually happens (OS allows all interrupts), but since RAX does not want it, it is stacked. Some time later, RAX again starts running. Soon it normally changes its system mask, expecting any pending interrupts to occur. Now however, there are none. The Hypervisor has one stacked, but if it does not know about the change in the

RAX system mask, it can do nothing. The answer is, at these critical times, to ask the Hypervisor if it has any pending interrupts to pass on to RAX. The mechanism used is as follows. All critical instructions in RAX which can change the system mask are replaced by specific, invalid, S/360 operation codes. When these are executed, they cause a PROGRAM CHECK to occur specifying an invalid operation code. The program check handler contains programming to recognize these special "errors". On finding one, it goes to the Hypervisor to check if any pending interrupts are allowed by the new system mask. If none are found, the program check routine goes back to the program (after setting the system mask as the original instruction would have done). If an interrupt is found, PSWs in RAX's low core are set up to make it look as if the normal instruction had been executed, and immediately, the pending interrupt had occurred. To RAX this looks identical to what would have happened on the real machine.

4. Several other small changes exist. There is a routine to get the current CSW (Fig. 2) and return it to the Hypervisor. The Hypervisor, when it wants to stack an I/O interrupt for RAX must save the CSW, but as the channel stores it in RAX's core box, the Hypervisor (residing in the OS section), cannot get it itself.

The RAX console handler recognizes console commands starting with "==" and passes them on to the Hypervisor. It also allows for a reply from the Hypervisor to be printed.

It should be noted that all these changes only take effect when the Hypervisor IPLs RAX. All these modifications contains checks to make sure that if RAX is running as a stand-alone system, no branch is ever made to the various Hypervisor routines (which do not exist)!

CHAPTER 8

HYPERVISOR PERFORMANCE

The main purpose of the Hypervisor as described here is to allow RAX to operate on an IBM 360 Model 65 unencumbered, while allowing a S/360 operating system to use any CPU time that is left over. The implementation just detailed seems to do this quite well.

Tests were run to obtain some measure of how well the Hypervisor works and to find out how different OS job types were affected by running under the Hypervisor. The general approach was to process a certain job (or jobs) using OS/360 without RAX running, and observe their performance. The same jobs were also run while RAX was operating (during production hours- i.e. a loaded system). During this run, the performance of the OS jobs was again monitored. Also, some statistics from RAX with respect to its CPU usage were obtained.

Ideally, these tests should have been run in a carefully controlled, reproducible manner. Unfortunately this was not possible since it would have required a dedicated machine for a relatively long stretch of time. Also, at least thirty people at terminals (or another

computer to simulate them) would have been needed to place a realistic load on the RAX machine. Neither of the above machine configurations nor the people were at the disposal of the author. This necessitated performing most of the tests in a 'live' environment during normal production time. Several restrictions as to the exact kind of tests run and the statistics available from them were thus introduced.

The first problem was that no control whatever could be exercised over exactly how busy the RAX system was during the tests, or just what kind of work it was doing. Care was taken to ensure that RAX was not virtually idle, or completely saturated while most of the tests were run. Little could be done however to ensure that erratic things (such as sudden RAX compute bound jobs being run, or sudden lulls) did not occasionally occur.

Another bottleneck was that it was often hard to make the sampling of the RAX statistics coincide exactly with the start and end of the OS test runs. However, as the test runs lasted as much as thirty minutes, and the error in fetching the RAX statistics was not usually more than several seconds, the error is hopefully minimal. Extrapolation allows a quite accurate figure to be estimated.

The set of test jobs took about an hour to run from start to finish. Since these tests had to be run on the machine at Illinois Bell Telephone, which was heavily utilized, the number of test runs was naturally limited. Many of the tests had to be run while RAX was active, but these prime day-time hours were the time when they could least afford to give up the OS side of their machine for any length of time. Nevertheless, it is felt that enough tests were performed to give a good picture of the Hypervisor performance. As will be seen later, despite all these problems, the tests do seem to show that the RAX/OS Hypervisor does behave as an intuitive approach would indicate.

There are several effects which will tend to alter how much work OS can get done in a given period of time. The parameters which affect OS/360 are, the amount of CPU time it is given, and how this time is partitioned. If OS gets on the average, x percent of the total CPU time, a compute bound job normally taking t seconds will now run in (100/x)*t seconds, with an apparent time with respect to OS, of t seconds. If a very I/O bound job with a stand-alone elapsed time of t seconds is run, and it can get the CPU from RAX whenever necessary to restart I/O, the job will take only t seconds of real time with an apparent time to OS of anywhere from (x/100)*t to t seconds. In this case, the

apparent time to OS will depend upon how much of the I/O time is overlapped with RAX CPU work. It can be seen that in these simple cases, the apparent time to OS will never exceed t. In the case of real jobs, however, the RAX/OS interaction can increase apparent job times in addition to real clock time. This can be particularly important for jobs performing much disk input/output.

Consider the case of OS wanting to initiate a disk read for a record that is just about to pass under the read head of the drive. If the CPU is taken from OS before the request can be initiated, and RAX runs for long enough for this record to pass by, OS will now have to wait a full revolution of the disk (1/60th second) before its read request can be completed. Of course this same type of happening can lessen the apparent time of a disk request to OS by delaying action so that the request happens to complete sooner. The effect and frequency of this type of occurrence will be highly dependent on the job characteristics of both the OS and RAX tasks.

It must be remembered that the RAX job mix at Illinois Bell tended to be highly I/O bound (averaging only 25% CPU utilization). That is, much of the time, it is initiating I/O requests and attempting to go into wait state (i.e. go to OS/360). As soon as the I/O completes, it takes

control back from OS. The OS/360 job mix is also similarly I/O bound. As a result, it is quite likely that OS, during its time slot, will initiate an I/O request of its own and place the machine in a real wait state. Whichever request completes first, will cause control to be given to the appropriate supervisor. The devices (and thus their timings) for the two systems are basically the same. That is, one can expect the typical I/O request on either system to be of the same duration. Thus, if the situation just described is a correct picture, it is not unreasonable to expect OS to gain control often enough (at small enough intervals) to maintain a high I/O activity rate. This is true, except when the RAX machine goes into a solid CPU bound activity, completely locking OS out. The RAX job mix at Illinois Bell on the Model 65 tended to be either highly I/O bound, or completely CPU bound (momentarily at least). During the I/O times, OS should be able to do almost as much work as on a bare machine, and during the CPU time, it should do nothing. Thus, the overall efficiency of the OS/360 machine should be approximately equal to the percentage of I/O bound time (non CPU time) of the RAX machine.

The basic performance bench marks consisted of a set of OS/360 jobs. These were Fortran-G and Cobol jobs. The stream was largely I/O bound, although sections were

compute bound. It was felt that they would closely resemble the type of jobs normally run on the machine while RAX was active. Two versions of the job stream were run. The second version was different from the first in that it allowed the jobs to allocate more buffer memory than the others. This allowed I/O requests to be blocked and thus tended to make the stream slightly less I/O bound. Each stream was run once on a machine without RAX present, and twice while RAX was active.

The method used to determine the effects of running these jobs under the Hypervisor was to note the time of day (both real TOD and TOD according to the OS clock) at the start of the first job and also at the end of the stream. When RAX was not running, these two elapsed times were of course the same. When RAX was active, the OS clock only runs when RAX is not actively using the CPU and so it reflects only the time that OS has control of the whole machine.

The results in all cases were similar (Table III). The clock time to run the jobs under RAX was always greater than the time needed on a stand-alone system. The amount of extra time varied depending on how busy RAX was. However, the OS elapsed time was always less than on a stand-alone machine (the OS timer does not run while RAX is doing

TABLE III

OS/360 BENCH MARK TEST RESULTS

JOB STREAM	MODE	ELAPSED TIME	OS/360 TIME (RAX WAIT)	RAX CPU TIME	OS/360 EFFICIENCY	RAX CPU	RAX WAIT
1	S/A	1464	1464	-	100%	-	-
1	HYP	1803	1376	427	81%	24%	76%
1	HYP	1760	1366	394	83%	22%	78%
2	S/A	1176	1176	-	100%	-	-
2	HYP	1745	1123	622	67%	36%	64%
2	HYP	1593	1104	489	74%	31%	69%

All times are in seconds.

computing). Therefore, as far as OS/360 was concerned, the jobs ran faster. Using the first set of tests as an example, the jobs took 1464 seconds to run on the bare machine. The time under the Hypervisor and RAX was 1803 seconds. During the second run, the OS clock increased by only 1376 seconds. That is, during the 1803 seconds of real time, RAX was running as if it were alone on a S/360 model 65 CPU. Simultaneously, the OS side was performing 1464 seconds worth of work. In the surplus time discarded by RAX, the OS system was working at the rate of (1464/1803)*100% of that possible on a stand alone machine. This produces an efficiency of 81% for the OS/360 side of the machine. The extra 81% is not quite free of course. Additional hardware is needed to support this configuration, but it only increases the system cost by about 12%.

It should be noted that the RAX CPU utilization while the second set of jobs were being processed was appreciably higher than during the first set of jobs. This had nothing to do with the OS job stream characteristics. The RAX system was simply more heavily loaded (with compute bound jobs) during the latter tests.

As expected, the efficiency of the OS/360 partition is roughly the same as the percentage of wait time available to it. If anything, it is slightly higher, indicating that

it is taking advantage of RAX CPU time to allow I/O to complete (essentially "free" time).

Another type of test was also run. This program wrote records on a magnetic tape while concurrently performing CPU bound work. It was designed to find the maximum amount of CPU work that could be done without slowing down the tape I/O. That is, it was to find the maximum number of times it could execute a given arithmetic operation without increasing the elapsed time needed to write 100 records on tape. This test was done for a variety of record lengths. On a stand-alone system, the test gives the expected results; the number of loops varies linearly with the tape record size. When run under the control of the Hypervisor, it was hoped that the test would give some insight into the type of interference that the RAX system caused.

The results certainly did this, but not quite in the analytical way expected. The type of interference found could best be described as 'erratic'. Some of the tests, which happened to be run during a quiet spell on RAX, were virtually indistinguishable from those run on a bare machine. Another set not only showed completely different results, but showed that the program logic was not really able to cope with the situation it encountered. It became

quite confused when the RAX machine would alternate between being completely compute bound, and then almost completely idle. This fluctuation (between CPU and I/O bound work) happened too slowly to average itself out over a period of several seconds, and too fast to allow a complete run of the test to complete (20 seconds or so) without seeing some fluctuations. When running without RAX, the program could reliably use the logic that if, for example, 50 CPU loops during a I/O request saturated the computer, then certainly 100 loops would also. This was no longer true under the RAX-influenced Hypervisor. The bench mark tests produced reasonable results in this same environment because they average out all the system interactions over a period of 20-30 minutes. The tape test program, tries to compare results on a second by second basis, and this is simply not practical. The program could have been modified to average performance over a longer period of time, but then its overall running time would have been too long to execute during the available test period.

An important contributing factor to overall Hypervisor performance was the degradation to either or both operating systems due to the Hypervisor itself. If the Hypervisor, in performing its switching task was using too much of the CPU resources, it could never be a success. In order to estimate how much CPU time was being used by the

Hypervisor itself in performing its functions, a number of counts were kept within the Hypervisor module. These counts produced a record of exactly how many times all the key routines within the Hypervisor were entered, and the decision paths followed within these routines. The counts could be printed at any time by means of a console command. From the counts, it was determined which sections of the Hypervisor code were executed with some regularity. It must be remembered that many of the routines were programmed to handle 'special' case occurrences. It was not originally known exactly how often (if ever) these things happened. The counts showed that, in fact, all these special cases did happen at some time or another. Some counts were as high as 400,000 per hour of operation.

Execution times for most sections of the Hypervisor were calculated using the values for instruction execution times published by IBM.[8] Counts for a typical period of time were analyzed and combined with the times calculated above. For the 7 1/2 hour period sampled, the Hypervisor coding evaluated would have taken 171 seconds to execute. This works out to be only about 25 seconds per hour. Even if a generous allowance is made for sections of coding not included in this analysis (not often executed, but perhaps longer in length), the overhead due to the Hypervisor is still likely to be under 1%. This figure is quite low. It

is especially good when it is remembered that the CP/67 system mentioned earlier as possible alternative to the Hypervisor, typically had overheads of 30-50% in this type of environment.

The overhead attributable to maintaining the counts themselves was calculated to be 27 seconds during the same 7 1/2 hour period, or only 4 seconds per hour.

CHAPTER 9

OVERALL RESULTS AND THE FUTURE OF THE HYPERVISOR

All the essential requirements set forth in Chapter 2 were satisfied by the RAX-OS/360 Hypervisor. In addition, the Hypervisor met virtually all the criteria discussed in Chapter 4. Perhaps even more important, it has been found to be truly usable in a real production environment. Once the systems are IPL'd (a simple task), the Hypervisor is virtually invisible to the operations staff. They are running normal OS/360 and RAX systems. The RAX users rarely notice any difference from running on a bare machine.† The OS machine processes a reasonable number of both production and test jobs during the day. The Hypervisor, at Illinois Bell Telephone, is quite certainly a success.

The only major disadvantage of the Hypervised system is that the user is tied down to a particular piece of equipment. The Hypervisor feature is available only on the S/360 Model 50 and 65 CPUs. If the capacity of the Model 65 were to be reached, there is no machine to upgrade to (if the Hypervised approach is to be maintained). A

†The only real impact upon the RAX user was that the overall system failure rate was higher due to OS/360 crashes bringing RAX down with it.

possible solution will be brought forth, but first it is relevant to review several new computer models that IBM has made available since this project was first undertaken.

The System/370 line of IBM computers is a series of central processors and peripherals designed to be upward compatible with the S/360. That is, most S/360 programs (except very CPU model dependent ones - like the Hypervisor), will run on the new machines without modification. The new models offer certain economies over the older ones due to technological advances. In addition, they possess several facilities and features not normally found on S/360 computers.

Relocation memory is standard on most S/370 models. It was formerly available only on the S/360 Model 67. It has been argued that the ability to dynamically relocate programs within main storage is a valuable asset to a well designed time sharing system. However, a time sharing system, which offers the economies, versatilities, and OS/360 compatibilities of RAX was still not available from IBM. This fact has been emphasized by the fact that IBM has recently acquired the rights to market (on a royalty basis) the McGill-RAX operating system.† RAX would appear to still

†IBM is currently marketting the system under the name MUSIC (McGill University System for Interactive Computing).

be the best time sharing system running on IBM machines for use in many medium to large installations. However, as was the case with Illinois Bell, many of these same installation have a need to be able to run either OS/360 or its newly announced successor OS/VS2. A follow-on to the CP/67 system, VM/370 is available. It does allow several operating systems to be run on the same machine. It would seem that its performance is far better that was CP/67, however the overhead needed to multiprogram RAX and OS would still appear to be in the 30% range.

No Hypervisor function is currently available from IBM for any of the S/370 computers. However, several features of the S/370s make the concept very attractive. The new central processors have, in addition to the interval timer, several other means of maintaining both time of day and elapsed time. These new timers do not reside in main storage, and this would eliminate many of the problems experienced with the Model 65 Hypervisor implementation. The timers are designed to be used in multiprogramming systems and so would be better suited for use in the Hypervisor (a multisystem system) than the older interval timer. Secondly, and perhaps most important, all the S/370 processors currently available are highly microprogrammed machines.[9] That is, assembler instructions coded by programmers are not executed directly by the electronics of

the CPU, but rather are interpretively executed by a program written in a much more primitive language. This more basic instruction set varies with each machine, according to its internal design. The set of basic instruction needed to perform the S/370 operations are called the microprogram, and reside in a special storage within the CPU.

Most of the S/360 models were microprogrammed also. However, the newer machines tend to have two unique characteristics. The microprogram can be changed easily (sometimes even under the S/370 program control). This is as opposed to the S/360 implementation where a complicated engineering and/or manufacturing process was often needed to change the microprogram.

Secondly, the type of microprogram[10] used within the S/370s tends to lend itself to making relatively small modifications to the way in which instruction are performed. It is also easy (relatively speaking) to add special new machine functions. All this can often be done without any hardware (electronics or wiring) change to the computer. It would seem that the functional characteristics changes to most S/370s to implement a Hypervisor could be accomplished relatively easily and inexpensively (by IBM engineers at least) by means of microprogram changes. Such changes could even be designed to enable the Hypervisor to perform certain

of its functions very efficiently by providing specially designed CPU operations.

Lastly, it is worth noting that the parts of the Model 65 configuration required by the Hypervisor over and above a 'normal' configuration, were extra memory and channels. In the S/370 series, both of these have tended to come down in price. The new machines tend to be priced so that the bulk of the cost is for the CPU itself. If adding some extra memory and channels yields an 75% increase in production, (as it seems to have done at Illinois Bell), then it is indeed a good bargain.

The McGill University Computing Centre is currently operating an IBM S/360 Model 75 computer system running OS/360, and a S/370 Model 155 running MUSIC (RAX). The Model 75 is heavily loaded. The next larger machine in the current IBM product line is a S/370 Model 168. The Model 168 is considerably faster than the Model 75 and it also provides facilities (such as memory relocation hardware) which make it a very attractive machine. However, it is also considerably more expensive than the Model 75. If MUSIC could be run on this same machine, it would mean that the Model 155 would no longer be needed. This cost saving would help to finance the Model 168.

There are several ways of running MUSIC on the same machine as OS. One is to use VM/370. This would easily provide the function, but the estimated cost in terms of overhead seems to be very high (currently over 50% with OS/VS2). Perhaps in a few years, when VM/370 becomes more efficient, this could be a reasonable approach. Another way is to modify MUSIC so as to run as a program under OS instead of as a dedicated system. This is possible, but it is felt that to do so would remove most of the efficiencies that are among MUSIC's best assets.

The most promising method seems to be to Hypervise the two systems. IBM is currently considering a McGill request to provide a Hypervisor shared storage feature for the S/370 Model 168.

CHAPTER 10

CONCLUSIONS

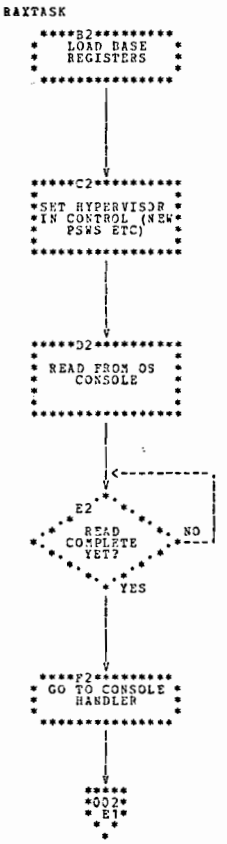
The McGill-RAX - OS/360 Hypervisor was proposed as a vehicle for providing a wide range of computing services using a single computer. Since its installation in 1970, the Hypervisor has proven to be extremely practical, efficient and reliable.

The new IBM System/370 line of computers contain many desirable features. Several of these would make it very much easier to write a Hypervisor program, others would enable the Hypervisor to be more sophisticated and general than the one described in this paper. It is reasonable to expect that the overhead of this expanded Hypervisor would still be in the order of 1-3%. All that is needed is for the Hypervisor hardware feature to become available for large scale System/370 computers.

APPENDIX A

HYPERVISOR FLOWCHARTS

INITIALIZATION



HYPERVISOR INITIALIZATION ALLOWS THE THE OPERATOR TO START EITHER BAX OR OS/360.

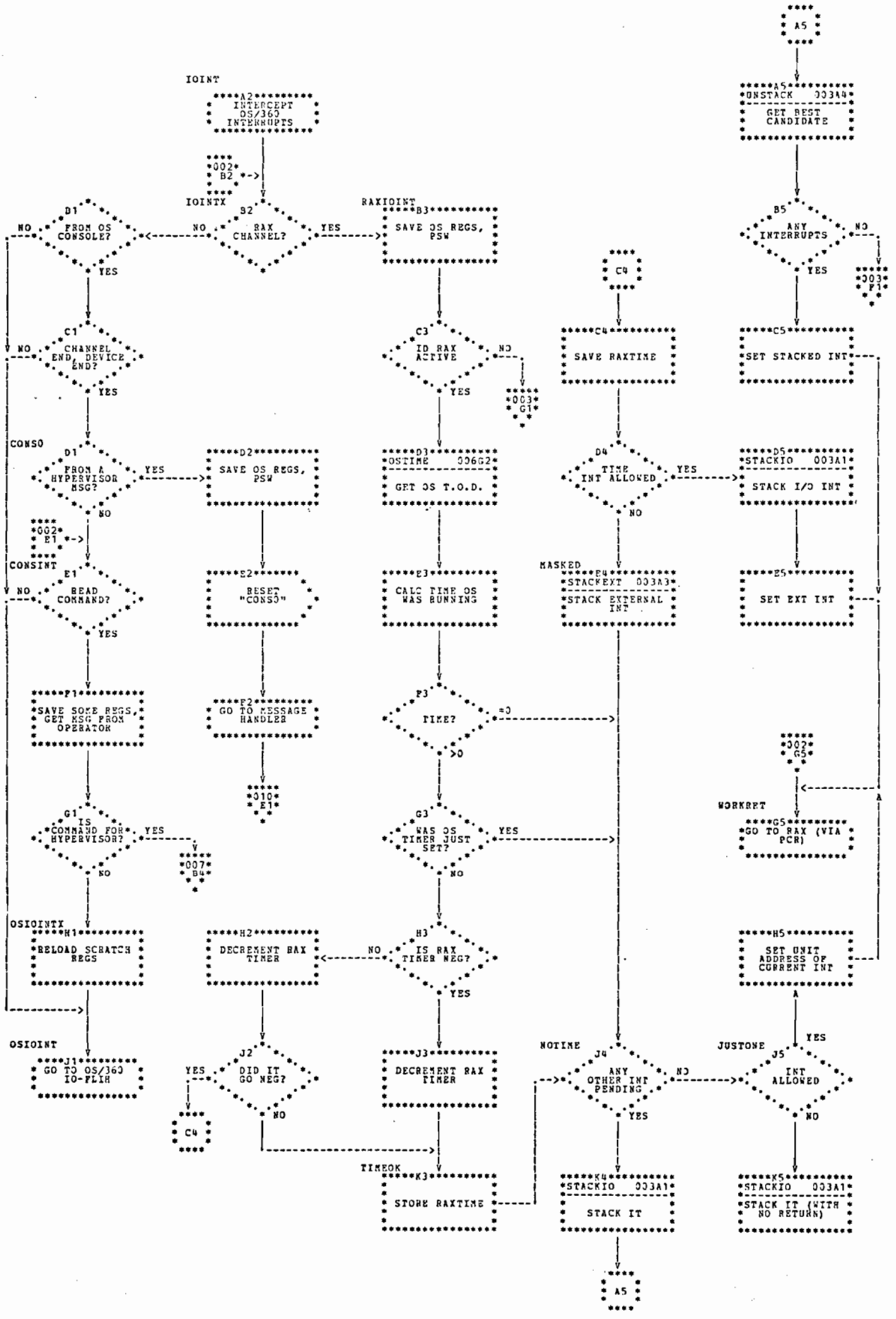
WHEN BAX IS STARTED, CONTROL IS RETURNED TO THE HYPERVISOR VIA THE "WAIT" ROUTINE.

WHEN OS/360 IS STARTED, CONTROL IS RETURNED VIA "IOCATCH" TO PATCH LOW-CORE SO THAT THE HYPERVISOR RECIEVES ALL FURTHER I/O INTERRUPTS.

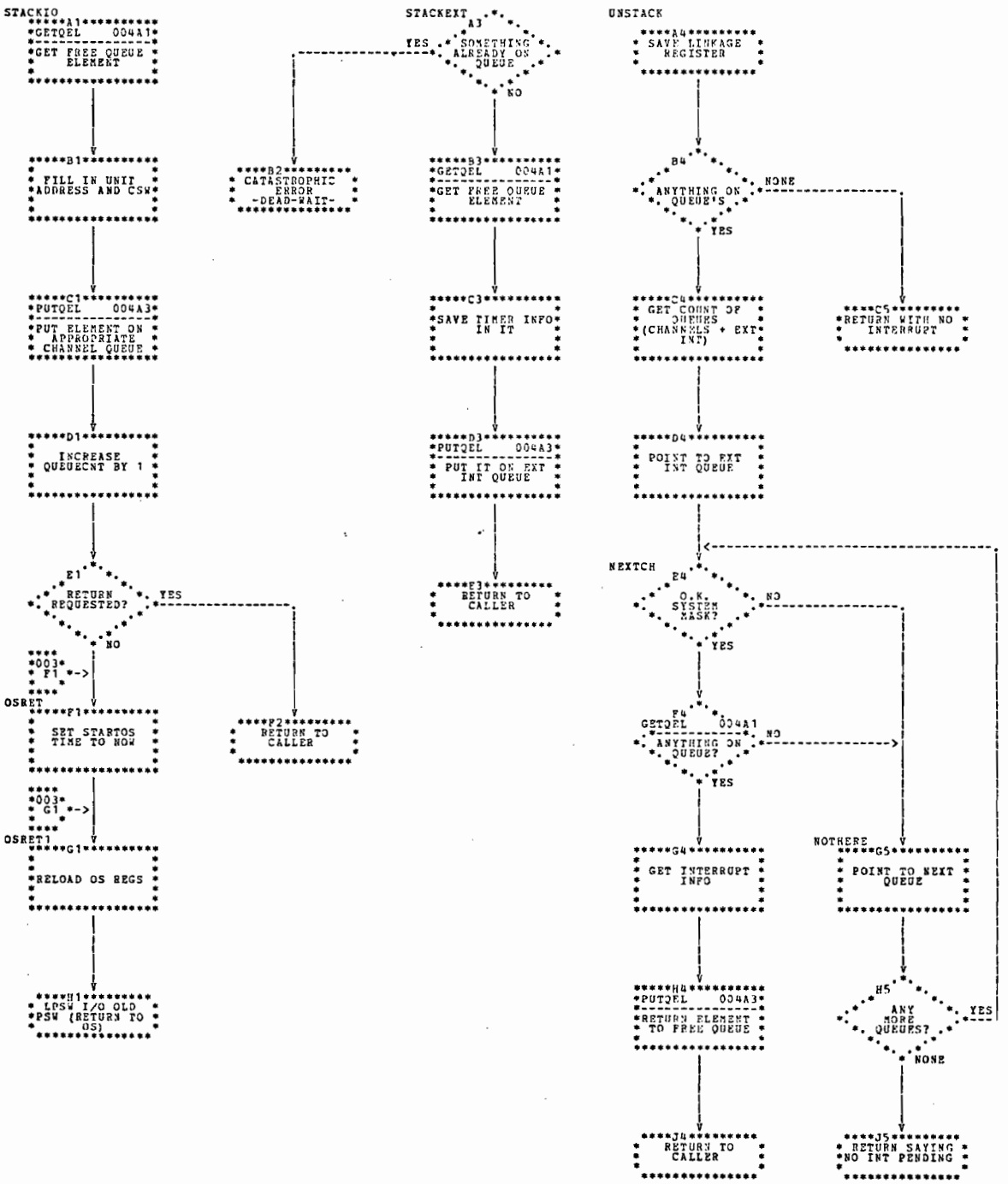
THE FIRST TIME OS GOES INTO WAIT STATE, "OSWAIT" IS ENTERED TO GIVE CONTROL OF THE MACHINE BACK TO THE HYPERVISOR (AND THUS BAX IF IT IS RUNNING).

WHEN EITHER SYSTEM IS UP AND RUNNING, IT'S CONSOLE MAY BE USED TO START AND STOP THE OTHER SYSTEM.

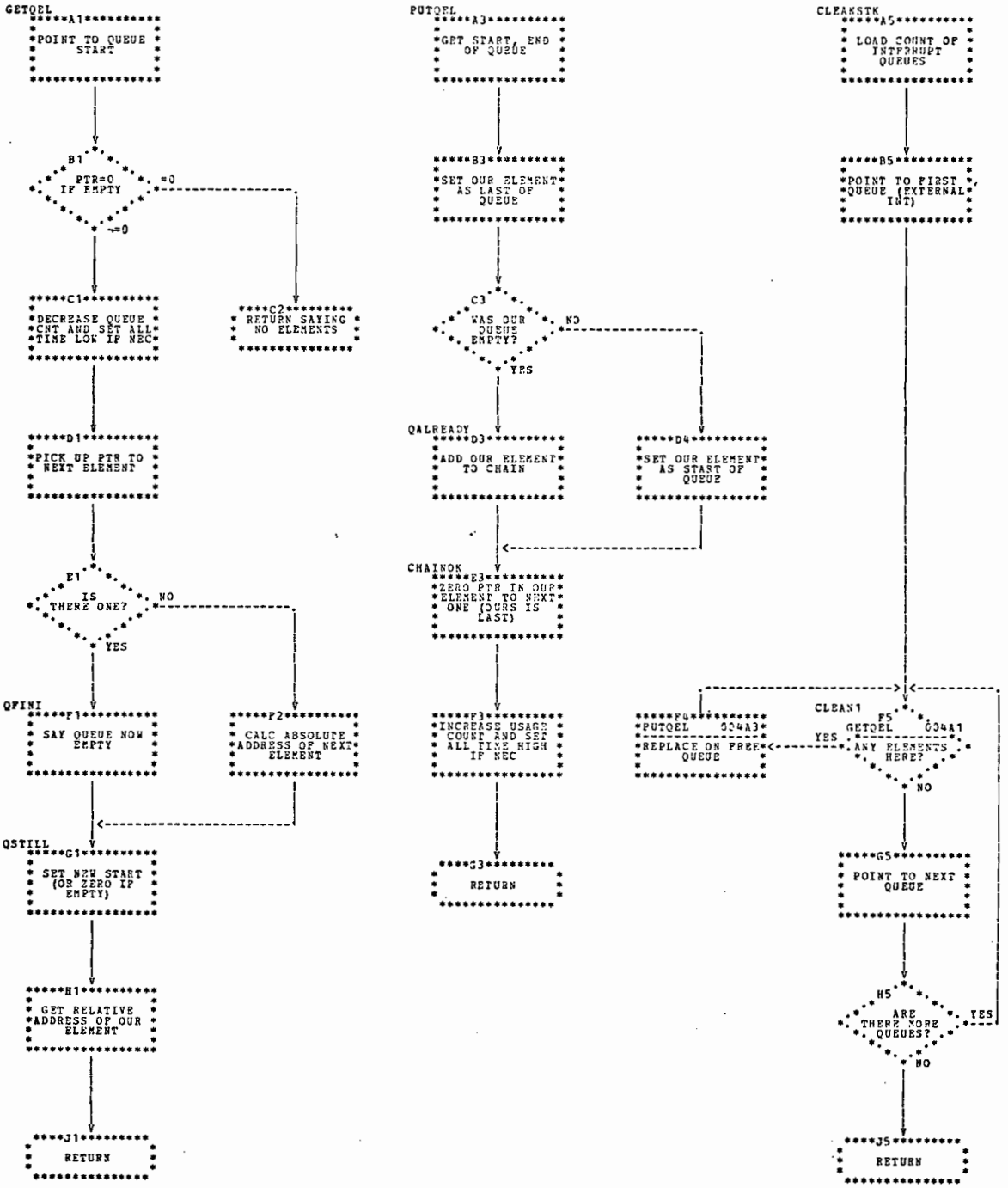
I/O INTERRUPT ROUTINE

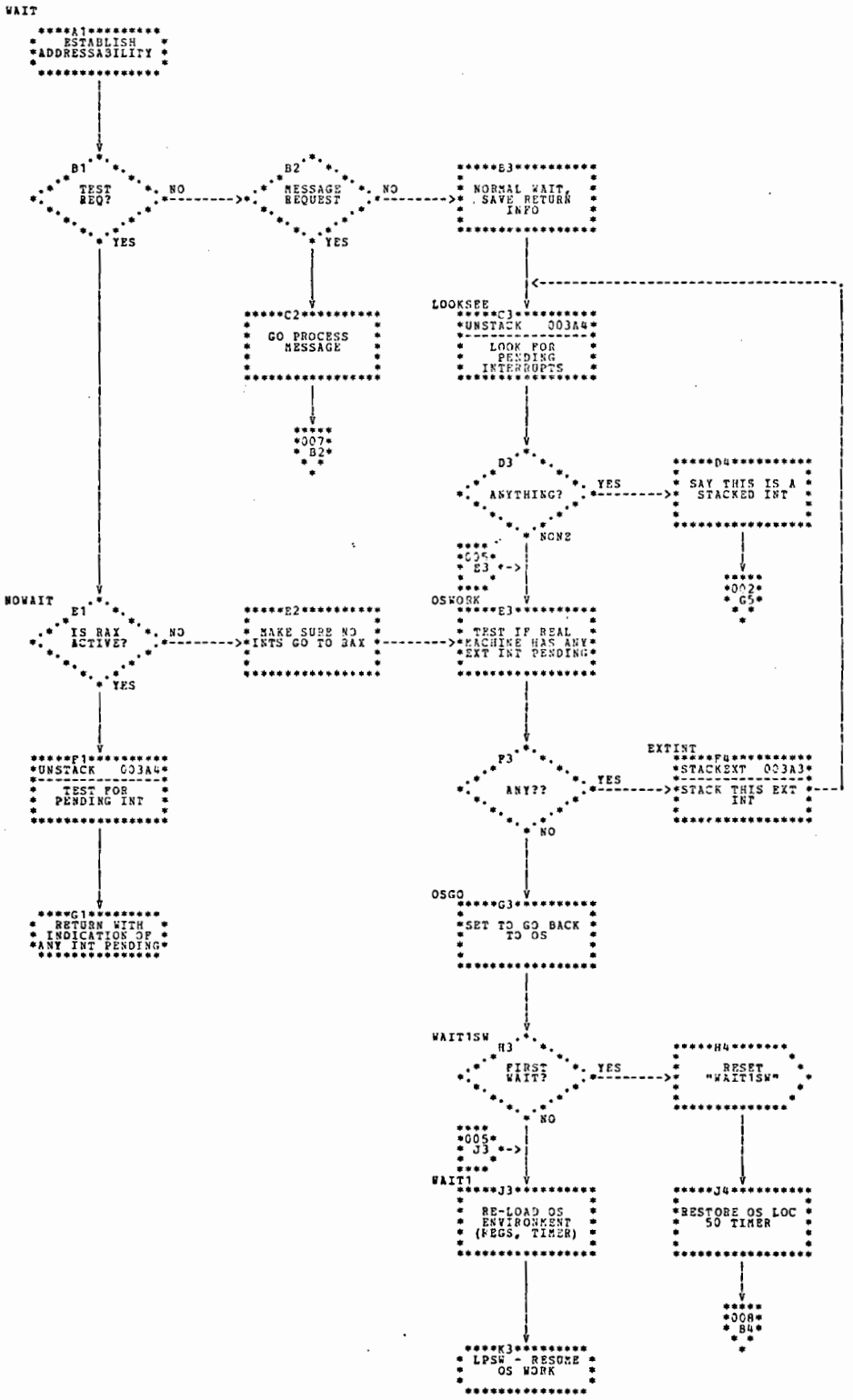


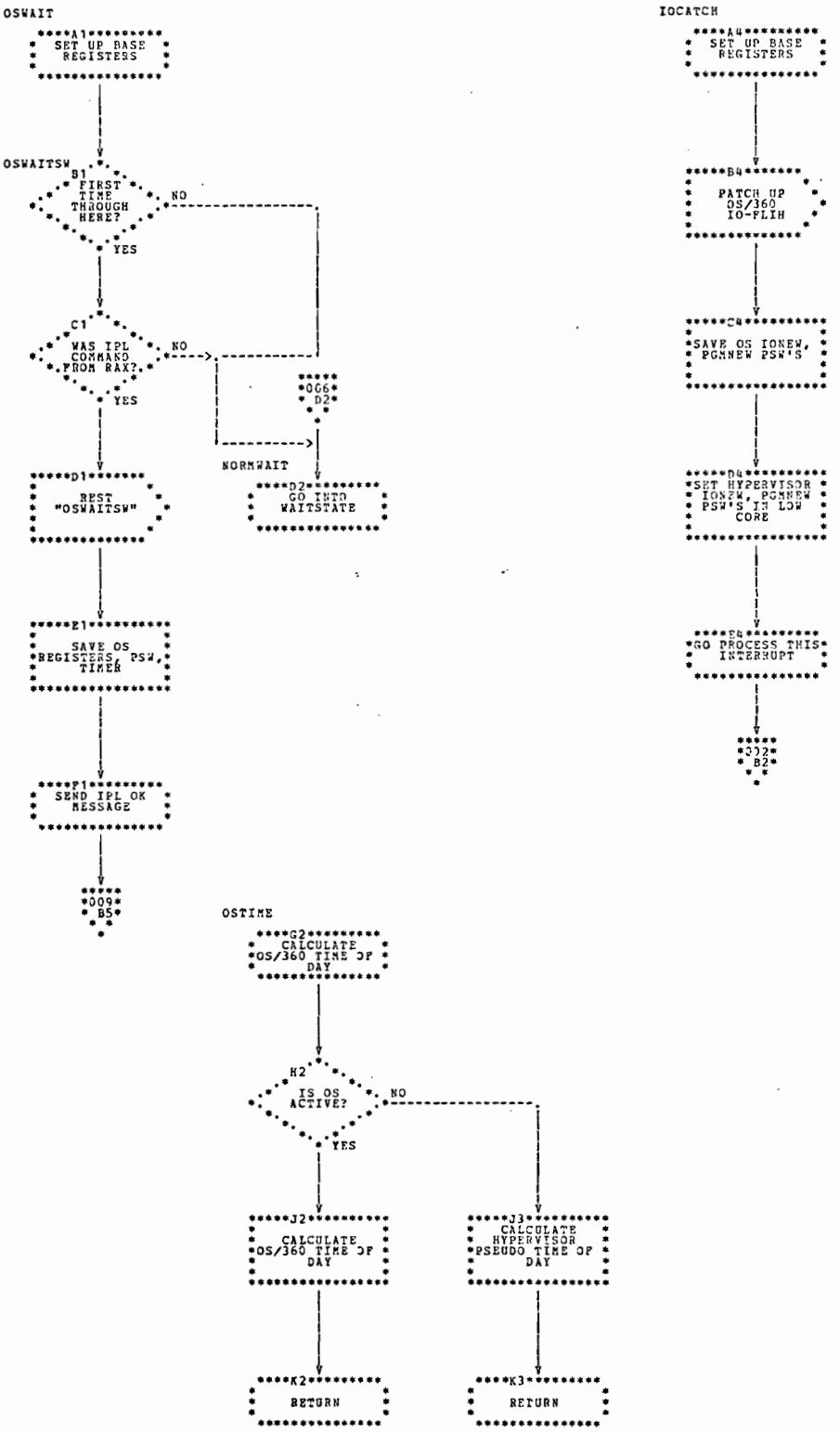
INTERRUPT QUEUE MANAGEMENT ROUTINES

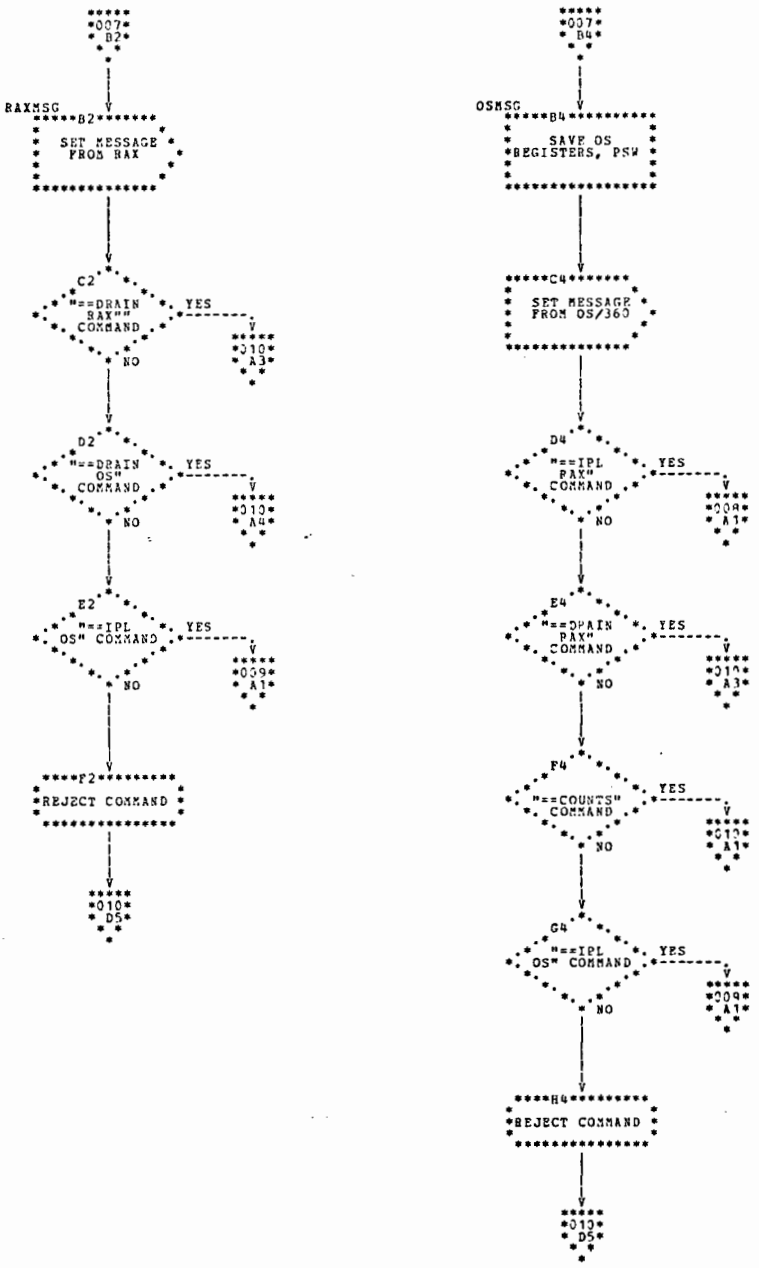


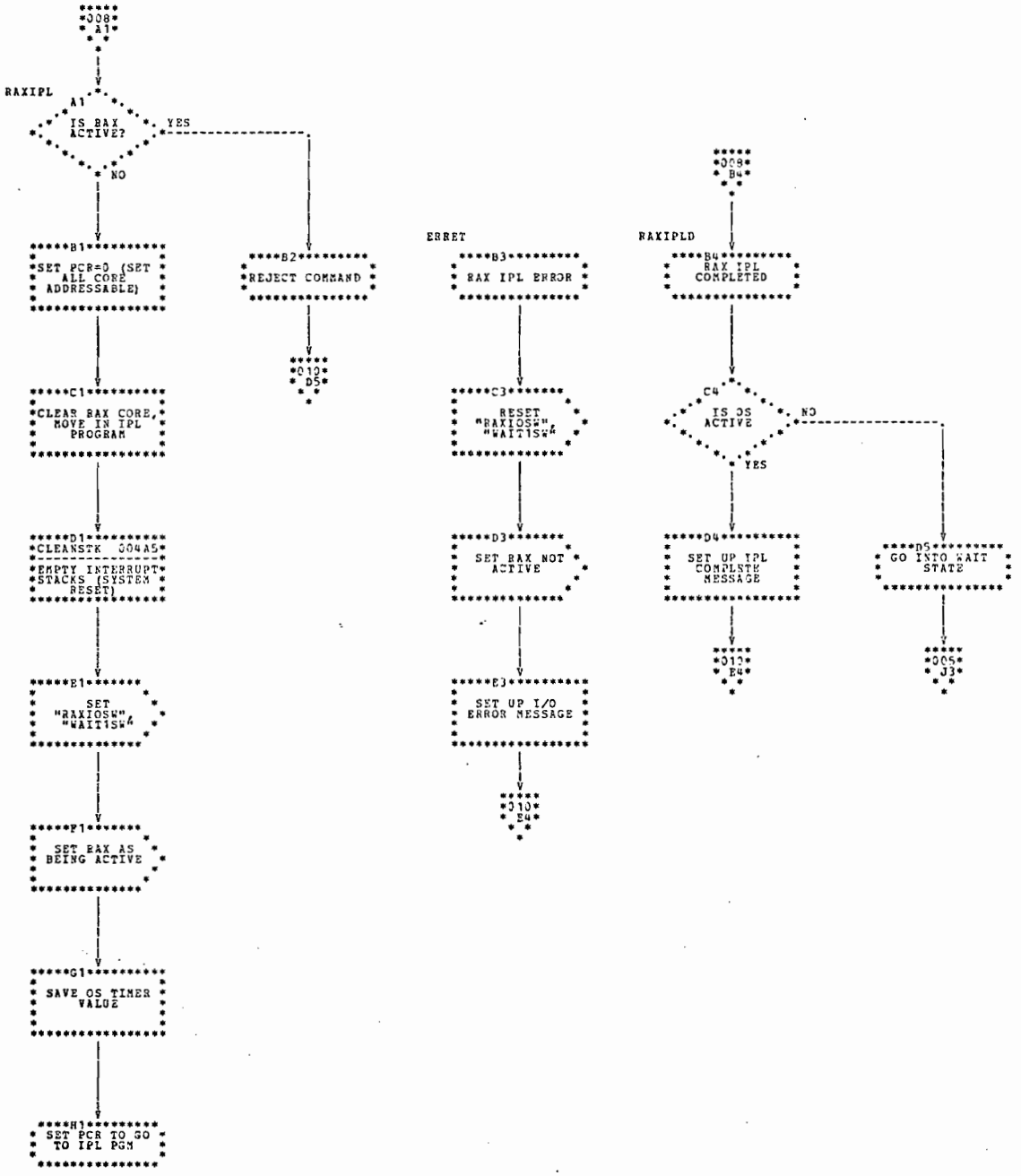
INTERRUPT QUEUE MANAGEMENT ROUTINES

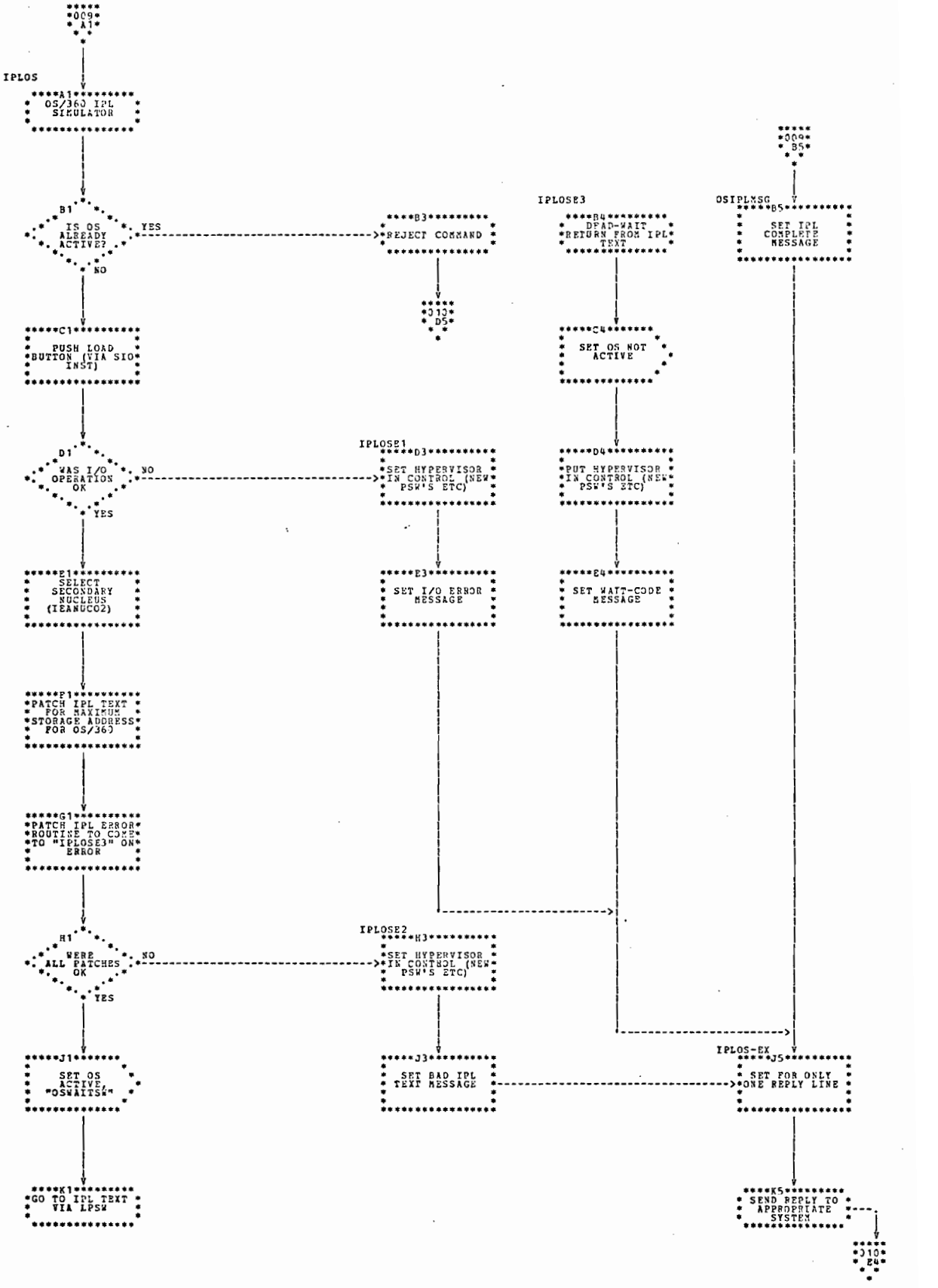


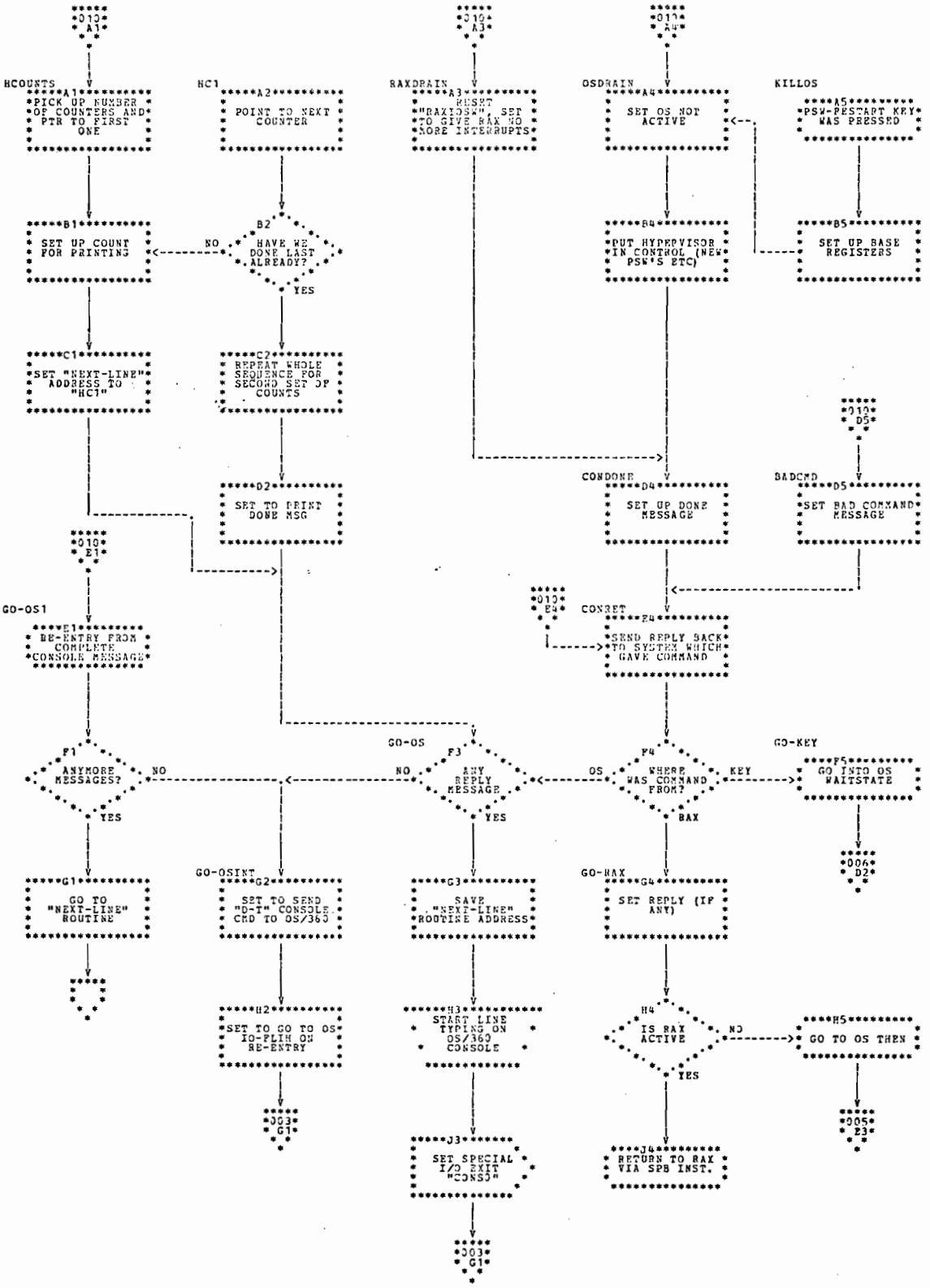

WAIT
A1
ESTABLISH ADDRESSABILITY
B1
TEST REQ?
NO
YES
B2
MESSAGE REQUEST
NO
YES
B3
NORMAL WAIT, SAVE RETURN INFO
C2
GO PROCESS MESSAGE
007 B2
LOOKSEE
C3
UNSTACK 003A4
LOOK FOR PENDING INTERRUPTS
D3
ANYTHING?
YES
NONE
D4
SAY THIS IS A STACKED INT
002 G5
005 E3
NOWAIT
E1
IS BAX ACTIVE?
NO
YES
E2
MAKE SURE NO INTS GO TO BAX
OSWORK
E3
TEST IF REAL MACHINE HAS ANY EXT INT PENDING
F1
UNSTACK 003A4
TEST FOR PENDING INT
F3
ANY??
YES
NO
EXTINT
F4
STACKEXT 003A3
STACK THIS EXT INT
G1
RETURN WITH INDICATION OF ANY INT PENDING
OSGO
G3
SET TO GO BACK TO OS
WAIT1SW
H3
FIRST WAIT?
YES
NO
H4
RESET "WAIT1SW"
005 J3
WAIT1
J3
RE-LOAD OS ENVIRONMENT (REGS, TIMER)
J4
RESTORE OS LOC 50 TIMER
008 B4
K3
LPSW - RESUME OS WORK


OSWAIT
A1
SET UP BASE REGISTERS
OSWAITSW
B1
FIRST TIME THROUGH HERE?
NO
YES
C1
WAS IPL COMMAND FROM RAX?
NO
YES
066
D2
NORMWAIT
D2
GO INTO WAITSTATE
D1
RESET "OSWAITSW"
E1
SAVE OS REGISTERS, PSW, TIMER
F1
SEND IPL OK MESSAGE
009
B5
IOCATCH
A4
SET UP BASE REGISTERS
B4
PATCH UP OS/360 IO-FLIH
C4
SAVE OS IONEW, PGMNEW PSW'S
D4
SET HYPERVISOR IONEW, PGMNEW PSW'S IN LOW CORE
E4
GO PROCESS THIS INTERRUPT
002
B2
OSTIME
G2
CALCULATE OS/360 TIME OF DAY
H2
IS OS ACTIVE?
NO
YES
J2
CALCULATE OS/360 TIME OF DAY
J3
CALCULATE HYPERVISOR PSEUDO TIME OF DAY
K2
RETURN
K3
RETURN


007
B2
BAXMSG
B2
SET MESSAGE FROM BAX
C2
"==DRAIN BAX"" COMMAND
YES
010
A3
NO
D2
"==DRAIN OS" COMMAND
YES
010
A4
NO
E2
"==IPL OS" COMMAND
YES
009
A1
NO
F2
REJECT COMMAND
010
D5
007
B4
OSMSG
B4
SAVE OS REGISTERS, PSW
C4
SET MESSAGE FROM OS/360
D4
"==IPL BAX" COMMAND
YES
009
A1
NO
E4
"==DRAIN BAX" COMMAND
YES
010
A3
NO
F4
"==COUNTS" COMMAND
YES
010
A1
NO
G4
"==IPL OS" COMMAND
YES
009
A1
NO
H4
REJECT COMMAND
010
D5


008
A1
RAXIPL
A1
IS RAX ACTIVE?
YES
NO
B1
SET PCR=0 (SET ALL CORE ADDRESSABLE)
B2
REJECT COMMAND
010
D5
C1
CLEAR RAX CORE, MOVE IN IPL PROGRAM
D1
CLEANSTK 004A5
EMPTY INTERRUPT STACKS (SYSTEM RESET)
E1
SET "RAXIOSW" "WAITISW"
F1
SET RAX AS BEING ACTIVE
G1
SAVE OS TIMER VALUE
H1
SET PCB TO GO TO IPL PGM
ERRET
B3
RAX IPL ERROR
C3
RESET "RAXIOSW" "WAITISW"
D3
SET RAX NOT ACTIVE
E3
SET UP I/O ERROR MESSAGE
010
E4
008
B4
RAXIPLD
B4
RAX IPL COMPLETED
C4
IS OS ACTIVE
NO
YES
D4
SET UP IPL COMPLETE MESSAGE
D5
GO INTO WAIT STATE
010
E4
005
J3


009 A1
IPLOS
A1 OS/360 IPL SIMULATOR
B1 IS OS ALREADY ACTIVE?
YES
NO
B3 REJECT COMMAND
D13 D5
C1 PUSH LOAD BUTTON (VIA SIO INST)
D1 WAS I/O OPERATION OK
NO
YES
IPLOSE1
D3 SET HYPERVISOR IN CONTROL (NEW PSW'S ETC)
E1 SELECT SECONDARY NUCLEUS (IEANUC02)
E3 SET I/O ERROR MESSAGE
F1 PATCH IPL TEXT FOR MAXIMUM STORAGE ADDRESS FOR OS/360
G1 PATCH IPL ERROR ROUTINE TO COME TO "IPLOSE3" ON ERROR
H1 WERE ALL PATCHES OK
NO
YES
IPLOSE2
H3 SET HYPERVISOR IN CONTROL (NEW PSW'S ETC)
J1 SET OS ACTIVE "OSWAITS4"
J3 SET BAD IPL TEXT MESSAGE
K1 GO TO IPL TEXT VIA LPS4
IPLOSE3
B4 DEAD-WAIT RETURN FROM IPL TEXT
C4 SET OS NOT ACTIVE
D4 PUT HYPERVISOR IN CONTROL (NEW PSW'S ETC)
E4 SET WAIT-CODE MESSAGE
009 B5
OSIPLMSG
B5 SET IPL COMPLETE MESSAGE
IPLOS-EX
J5 SET FOR ONLY ONE REPLY LINE
K5 SEND REPLY TO APPROPRIATE SYSTEM
D10 E4


010 A1
HCOUNTS
A1 PICK UP NUMBER OF COUNTERS AND PTR TO FIRST ONE
B1 SET UP COUNT FOR PRINTING
C1 SET "NEXT-LINE" ADDRESS TO "HC1"
HC1
A2 POINT TO NEXT COUNTER
B2 HAVE WE DONE LAST ALREADY?
NO
YES
C2 REPEAT WHOLE SEQUENCE FOR SECOND SET OF COUNTS
D2 SET TO PRINT DONE MSG
010 E1
GO-OS1
E1 RE-ENTRY FROM COMPLETE CONSOLE MESSAGE
F1 ANYMORE MESSAGES?
NO
YES
G1 GO TO "NEXT-LINE" ROUTINE
003 G1
010 A3
RAXDRAIN
A3 RESET "RAXJOB" SET TO GIVE RAX NO MORE INTERRUPTS
GO-OS
F3 ANY REPLY MESSAGE
NO
YES
GO-OSINT
G2 SET TO SEND "D-T" CONSOLE CMD TO OS/360
H2 SET TO GO TO OS IO-FLIH ON RE-ENTRY
003 G1
G3 SAVE "NEXT-LINE" ROUTINE ADDRESS
H3 START LINE TYPING ON OS/360 CONSOLE
J3 SET SPECIAL I/O EXIT "CONSJ"
003 G1
010 A4
OSDRAIN
A4 SET OS NOT ACTIVE
B4 PUT HYPERVISOR IN CONTROL (NEW PSW'S ETC)
KILLOS
A5 PSW-RESTART KEY WAS PRESSED
B5 SET UP BASE REGISTERS
010 D5
BADCMD
D5 SET BAD COMMAND MESSAGE
CONDONE
D4 SET UP DONE MESSAGE
010 E4
CONRET
E4 SEND REPLY BACK TO SYSTEM WHICH GAVE COMMAND
F4 WHERE WAS COMMAND FROM?
OS
KEY
RAX
GO-KEY
F5 GO INTO OS WAITSTATE
006 D2
GO-RAX
G4 SET REPLY (IF ANY)
H4 IS RAX ACTIVE
NO
YES
H5 GO TO OS THEN
005 E3
J4 RETURN TO RAX VIA SPB INST.

BIBLIOGRAPHY

1. R.F. Rosin, "Supervisory and Monitor Systems," *Computing Surveys*, Vol. 1, No. 1, 37-54, (March 1969).

2. *IBM System/360 Remote Access Computing System (RAX) Version 4 Program Description Manual*, GH20-0354.

3. P.J. Denning, "Virtual memory," *Computing Surveys*, Vol. 2, No. 3, 153-189, (September 1970).

   R.P. Parmalee, T.I. Peterson, C.C. Tillman, and D.J. Hatfield, "Virtual storage and virtual machine concepts," *IBM Systems Journal*, Vol. 11, No. 2, 99-130.

4. *IBM System/360 Principles of Operation*, GA22-6821.

5. J.M. Chiarello, *IBM Simultaneous Use of Operating Systems and/or Emulators under Hypervisor Shared Storage RPQ*, Z77-9086.

6. *IBM*, Internal paper - only identification "Appendix D".

7. *IBM System/360 Operating System MVT Supervisor Program Logic Manual*, GY28-6659

8. *IBM System/360 Model 65 Functional Characteristics*, GA22-6884.

9. S.S. Hussen, *Microprogramming: Principles and Practices*, Prentice-Hall, Englewood Cliffs, N.J., (1970).

10. *IBM An Introduction to Microprogramming*, GF20-0385