UNISYS

V Series MCP/VS Architecture

Manual

CUSTOMER SERVICE ENGINEERING CONFIDENTIAL AND PROPRIETARY DATA

Relative to Release
Level 2.0

July 1987
Distribution Code **EF**
Printed in U S America
5026289

PROPRIETARY NOTICE

Comments or suggestions regarding this document should be submitted on a Field Communication Form (FCF) with the CLASS specified as 1 (H.W:Hardware), and the Type specified as 3 (DOC), and the product specified as the 7-digit form number of the manual (for example, 5026289).

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

LIST OF ILLUSTRATIONS

LIST OF TABLES

ABOUT THIS MANUAL

MCP/VS 2.0 is an operating system developed for the Unisys V Series systems. This advanced operating system incorporates many concepts and features that are new to Unisys products. Increased system throughput, controlled sharing of program resources, increased system reliability, and the maintenance of object code compatibility with previous Unisys B 2000/B 3000/B 4000 Series systems are the most important design goals of the operating system.

The purpose of this document is to introduce field support personnel and systems programmers to the overall design of the operating system, while shielding the reader from a great deal of low-level detail.

ORGANIZATION OF THIS MANUAL

This book is organized as follows:

MEMORY ARCHITECTURE AND MANAGEMENT

The first four sections are concerned with the topic of memory architecture and management. Several new concepts are introduced in these sections relating to memory organization. Essentially, the memory architecture dictates the design principles for the remainder of the operating system architecture.

TASK EXECUTION IN THE NEW ENVIRONMENT

The next section, Task Execution, describes the user environment while user code is executing, or while MCP code is running for the user task.

The fact that multiple users can be in the operating system and simultaneously accessing the same MCP function code means that the MCP is a multi-threaded operating system. In an environment where multiple users have access to shared data structures, it is essential to protect those structures as well as the code and data belonging to the individual processes. How MCP/VS 2.0 maintains controlled sharing of program resources is covered in Section 6, Code and Data Protection.

SYSTEM ELEMENTS

The 2.0 release of MCP/VS is composed of many seperate modules that are linked together. These modules, such as the Memory Manager, Message Module, and Global Module are mentioned throughout the manual. Sections 7 and 8 cover two of the most important system elements: the Kernel (an operating system module) and Independent Runners (a group of operating system tasks.) The Kernel is covered in detail in Section 7.

Independent Runners are another essential part of the operating system. In essence, they are tasks, except that they run for the operating system instead of for a user task. They perform basic system operations such as housekeeping functions and I/O-related services. They are discussed in Section 8.

INTERRUPT HANDLING, FAULT HANDLING, AND TASK DISPATCHING

Sections 9 through 11 cover the flow of control during instruction stream execution. Interrupts and faults can occur during task execution. The Fault Handler is discussed in Section 10.

The Kernel performs two discrete functions related to the determination of the next task or instruction to be executed: interrupt handling and the dispatching of the next task to be run on the system. Interrupt Handling is discussed in Section 9.

The decision as to which task to run next is made by the Dispatcher, a part of the Kernel. It is covered in Section 11.

VIRTUAL I/O

Only the most primitive level of the operating system needs to be concerned with the low level details necessary in performing input/ouput operations (I/Os) on the various physical devices. The concentration of device-dependent information in the primitive layer of the MCP means that the rest of the operating system can communicate with the primitive level using virtual I/O. Any changes to devices, or addition of devices to the system requires that only the primitive layer be updated. Virtual I/O is covered in Section 12.

TASK CREATION

Several actions are required in order for a task to be created. These steps are taken in the interval between when the task execution is requested and when the first instruction in that task can be executed. Task creation is discussed in Section 13.

SYSTEM SUPPORT TOOLS

Finally, two appendixes are devoted to system support tools. Appendix A discusses the Debugger, and Appendix B covers both the new and the most important instructions in the instruction set. The purpose, format, and implementation of each instruction are discussed.

SECTION 1
CONCEPTUAL SYSTEM OVERVIEW

TASKS, ENVIRONMENTS, AND MEMORY AREAS

Conceptually, the MCP/VS operating environment consists of tasks, environments, and memory areas. The highest level of granularity is the task. Every user program operating on the system at a given time does so as a separate task. Additionally, many internal operating system functions, such as device status routines, are treated as separate tasks.

Each unique task has associated with it a Reinstate List entry and a Task Number. The entire Reinstate List (RL) is arranged as an array, and the particular RL entry for any task can be located by using the Task Number as an index to the array.

Presumably all tasks are completely independent of each other, so that if enough physical processors were available, all of the tasks could be executed simultaneously. In the real world, it is unlikely that there are enough processors available to run all tasks at once, and even if there were, the tasks would contend with each other for resources. Resultantly, some number of tasks would be waiting on a resource at any given time, and throughput would not be optimal.

The Reinstate List is used to maintain the current state of a task while it is waiting to be executed again. The Reinstate List entry for a task contains a pointer to the Environment Table (ET). The ET defines the operating environment for that particular task. A task can have several different environments, but it can be in only one of these environments at any given time.

An environment consists of the code and data required to perform some function. For instance, when the user task is executing, the operating environment consists of the memory occupied by the user program code and data. If the program executes a READ BCT, then the operating environment will consist of the MCP code for the READ function and those areas of memory that contain the tables and structures (IOATs and Disk File Headers, for example) required by the READ function. If the READ function determines that it must do a physical I/O operation, yet a third environment will be entered that will consist of the physical I/O code and the data structures it requires.

The various environments that a task has access to are defined by a structure known as an Environment Table. The Reinstate List entry for a task contains a pointer to the Environment Table for that task.

Every environment consists of several Memory Areas. At the least, every executable environment must have a Code Memory Area and a Data Memeory Area. As many as six additional memory areas can be accessible in the environment.

The memory areas accessible from a given environment are defined by a structure called a Memory Area Table. The Environment Table for a task is an array indexed by Environment Number. Each entry in this array has a pointer to a corresponding Memory Area Table.

SECTION 2
MEMORY ARCHITECTURE

CONCEPTUAL MEMORY MANAGEMENT

At any instant of time a program (or an MCP function operating for the program) can have access to up to eight million digits of memory space. The program can view this memory as contiguous space in a linear address range of 0 to 7,999,999 digits.

In actuality, this space consists of eight memory areas numbered 0 to 7. Each memory area has its own Base/Limit pair used for locating it in physical memory and determining the size of the area. These Base/Limit pairs are contained in an 8-element array known as the Memory Area Table (MAT). The MAT is located outside of the user's addressing space and therefore can not be modified by the program.

Which memory area to be used at a given time is determined by the millions digit of a memory address. Memory Area 0 is addressed as 0 to 999,999; Memory Area 1 is addressed as 1,000,000 to 1,999,999, and so on.

While memory areas might appear to be contiguous to a user, it is highly improbable that they would be allocated contiguously in physical memory. Thus, it is not possible to have any data structure that crosses a million-digit memory area boundary.

MEMORY AREAS

A memory area is an allocatable unit of memory. It is a collection of up to 1,000,000 contiguous and logically related digits.

A memory area can range in size from a minimum of 1,000 digits (1 KD) to a maximum of 1,000,000 digits (1 MD), with a granularity of 1 KD. For example, if a user's code is 7,450 digits in length, the MCP will assign it a memory area of 8,000 digits.

A program can have access to a maximum of 8 memory areas at any instant.

LOCATING A MEMORY AREA

A memory area is located by a pair of Base/Limit addresses contained in a Memory Area Table (MAT). A MAT contains the addressing environment for a task (that is, an array containing the Base/Limit pairs). For references to data or code in the local addressing environment, the base relative memory addresses are added to the selected Base value in order to determine the absolute memory location.

MEMORY ACCESS

There is no need for the operating system to access all of memory at a given instant in time. To see why this is, refer to Figure 2-1 where a user task is performing a Read function. The user task needs to address only the following areas of memory in order to perform the Read:

- the MCP Read function code,
- the user task's data area,
- MCP data that is relevant to this task and this request, and
- the MCP system tables.

The remainder of memory is not accessed for the Read operation.

The memory area structure allows the operating system to efficiently access portions of memory containing only the relevant code or data for the task.

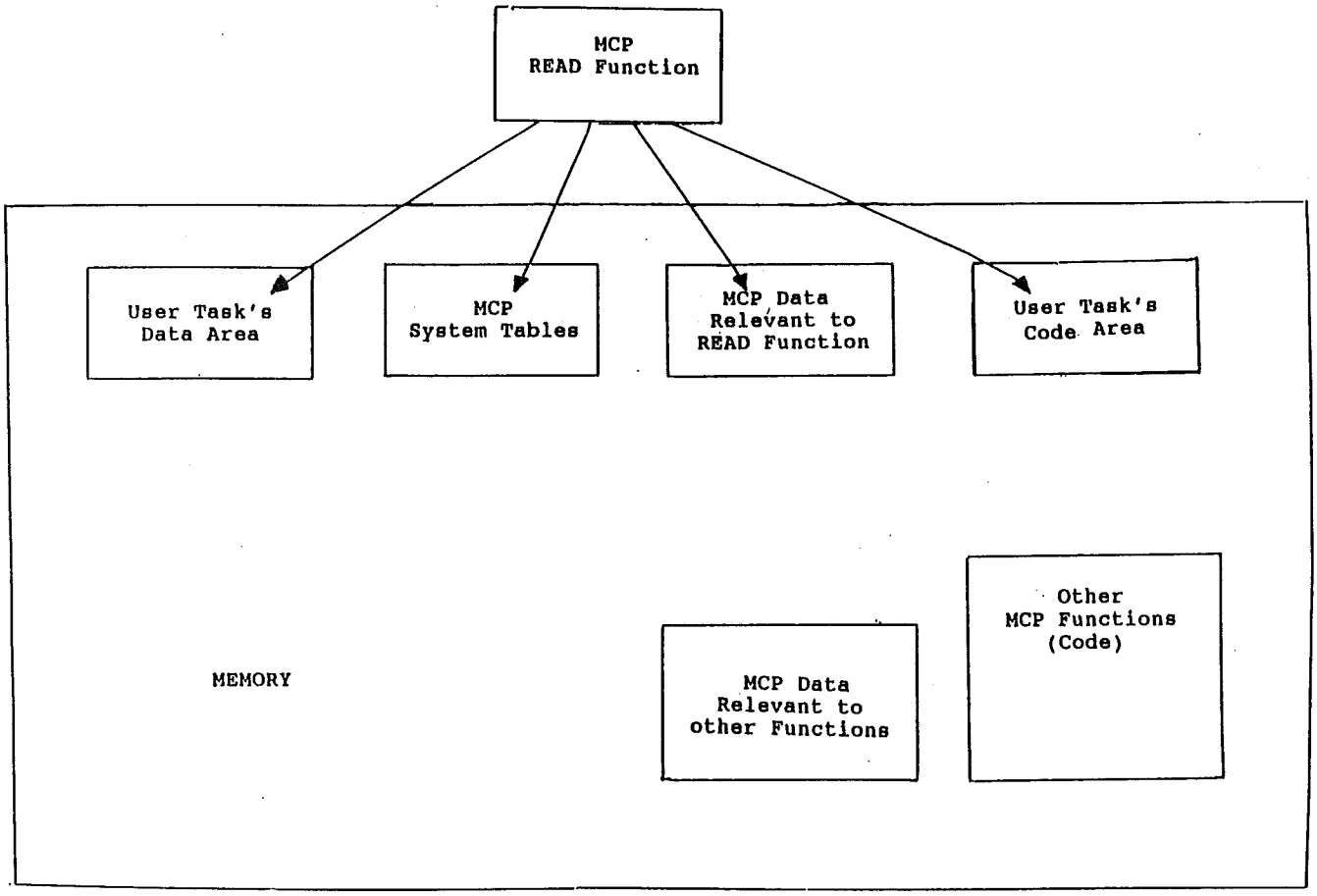

Figure 2-1. Not All of Memory Needs to be Addressed at Any Instant In Order to Perform a Function

INDEX REGISTERS

Three Index Registers (IX1, IX2, and IX3) occupy a reserved area of memory, relative to Base #0. They are located at memory addresses 08, 16, and 24, respectively.

These registers can be loaded with the Load Index Register instruction (LIX, OP=67) and stored with the Store Index Register instruction (SIX, OP=68). Additionally, any instruction that modifies memory can be used to save or modify the contents of these index registers. (Refer to Appendix B, LIX and SIX instructions.)

INDEX REGISTER FORMAT

The Index Register format is shown below. The first digit is the Sign Digit. Instead of a 7-digit offset, the new Index Register format has a 6-digit offset and a high order Base Selection Digit.

SN BSD 6-Digit Offset

BASE SELECTION DIGIT

The Base Selection Digit is used to select the Base/Limit register, or the memory area relative to which the 6-digit offset is to be resolved. The 6-digit offset is sufficient for referencing any digit within a 1MD memory area.

MOBILE INDEX REGISTERS

Four additional Index Registers have been defined, the Mobile Index Registers: IX4, IX5, IX6, and IX7. Unlike Index Registers 1, 2, and 3, which reside within program memory, IX4, IX5, IX6, and IX7 are external registers. They were created to ease the demand for Index registers, which are required to address Memory Areas 2 - 7.

These registers can be loaded individually or collectively with the Load Index Register instruction (LIX, OP=67) and stored individually or collectively with the Store Index Register instruction (SIX, OP=68). By specifying multiple 32-digit structures containing four 8-digit Index Register formats, it is possible to load and use these sets repeatedly. The net effect is the appearance of having an unlimited supply of Index Registers, while only having seven active at one time. (The LIX and SIX instructions are covered in Appendix B.)

These registers are fast and are generally used to point at frequently accessed structures, since they need be loaded only once. They are not stored in the user's addressing space and can only be loaded and stored with the LIX and SIX instructions.

The Index Registers are invoked using the indexed form of extended addresses with the extension digits of "D" and "E".

SECTION 3
ADDRESS RESOLUTION

GENERAL

Under the MCP for the V-Series machines, most of the MCP and all user programs are partitioned into a number of separate memory sections called memory areas. Each memory area is defined by its base and limit, which are always MOD 1000. All addresses to memory within this memory area are relative to one of these bases and limits.

All processes running under MCP on these machines have accessability to memory through eight Base/Limit pairs. Base #0, or Memory Area 0, is defined as that process' Data (context) Area. Base #1, or Memory Area 1, is defined as that process' code area.

Non-indexed addresses will refer to Base #0 or Base #1, depending on whether the address refers to the data or the code, respectively.

Processes can also address memory through any of the eight active Base/Limit pairs through the use of the Base Selection Digit in one of the Index Registers. With non-extended addressing, IX1, IX2, and IX3 can be used. Extended addressing also allows the use of IX4, IX5, IX6, and IX7.

INDEXED ADDRESSES

An address is interpreted as an offset from the base of some memory area. If the address is indexed, then the Index Register identifies which memory area. Its Base Selection Digit tells the processor to resolve the 6-digit offset relative to which Base/Limit register.

UNINDEXED ADDRESS RESOLUTION

If there is no Index Register on the operand, then the context of the address determines the memory area to which it is relative.

In general, all data references are relative to the current Data Area, and all code references are relative to the current Code Area. The Data Area is always Memory Area 0, and the Code Area is always Memory Area 1.

An Indirect Address reference is considered to be a data reference, and therefore the indirect address is assumed to reside in the Data Area, even if this is an indirect reference on a branch-type instruction. For those cases where constant data (such as long literals and branch tables) has been stored in the Code memory area, a special extended address controller is available to shift the context of the addressing from the Data Area to the Code Area.

INDIRECT ADDRESSES

The referenced address field can contain another address instead of the operand data. In turn, this address can point to another address. This indirect referencing can be carried to any depth. The controller of the final (and direct) address specifies the format of the operand field to be accessed and must conform to any address controller restrictions for that instruction.

Full generality of indexing is maintained in indirect addressing. Any or all of the indirect addresses in a chain can be indexed. An address is always indexed before the indirect reference is taken.

Extended addressing can be applied to any or all of the indirect addresses in a chain.

All indirect addresses in an indirect address chain that are context relative are relative to the Data Area (Base #0). If the indirect address is indexed, the specified Base Selection Digit from the Index Register is used to determine the base. If the indirect address is extended with a "D" and a Index Register is not specified, the address is relative to the Code Area (Base #1). (Refer to Table 3-1, Summary of Indexing and Address Resolution.)

RE-ENTRANT CODE

It is desirable to be able to have some constants in the Code Area that can be used as data. Thus, an unindexed extended address with the extension digit of "D" indicates that the address is relative to the Code Area, even if it is indirect.

This separation of code and data allows for re-entrant code. Data can reside in the code, but the data must be constant for the code to remain re-entrant.

Table 3-1. Summary of Indexing and Address Resolution

EXTENSION DIGIT	ADDRESS CONTROLLER (MOST SIGNIFICANT 2 BITS)			
	00XX	01XX	10XX	11XX
NONE	NO INDEX CONTEXT	IX1 BSD	IX2 BSD	IX3 BSD
A	ADDRESS ERROR	ADDRESS ERROR	ADDRESS ERROR	ADDRESS ERROR
B	ADDRESS ERROR	ADDRESS ERROR	ADDRESS ERROR	ADDRESS ERROR
C	NO INDEX CONTEXT	IX1 BSD	IX2 BSD	IX3 BSD
D	NO INDEX CODE BASE	IX4 BSD	IX5 BSD	IX6 BSD
E	ADDRESS ERROR	IX7 BSD	ADDRESS ERROR	ADDRESS ERROR
F	ADDRESS ERROR	ADDRESS ERROR	ADDRESS ERROR	ADDRESS ERROR

CONTEXT - CONTEXT RULES USED TO SELECT MEMORY AREA
CODE BASE - MEMORY AREA IS CODE AREA (MA 1)
BSD - MEMORY AREA IS EQUAL TO THE INDEX REGISTER'S BASE SELECTION DIGIT
ADDRESS ERROR - COMBINATION IS NOT VALID

SECTION 4
MEMORY MANAGEMENT

GENERAL

Memory is managed through the utilization of four memory resident tables:

- the Memory Area Status Table (MAST),
- many Memory Area Tables (MATs),
- many Environment Tables (ETs), and
- the Reinstate List (RL).

Together, these tables keep track of the memory areas and the addressing environment for a task and its subroutines. The firmware maintains an index pointing to the task's reinstatement information in the Reinstate List.

Each memory area currently defined on the system, regardless of which task or environment it is a part, has a corresponding entry in the Memory Area Status Table (MAST). Each MAST entry contains information on the size of the memory area, the Task Number, Environment Number, and Memory Area Number of the memory area corresponding to this MAST entry, and certain status information about the memory area. Refer to Table 4-2 for the MAST entry format.

A separate Memory Area Table (MAT) is assigned to each function, or environment, on the system. The entries in this table represent all memory areas claimed by the function at any given moment in time. Each MAT can contain a maximum of 100 entries.

There are two types of MATs for user tasks: the User Execute MAT (UEMAT) and the User Services MAT (USMAT). The UEMAT contains pointers to the memory areas that a user task has access to while it is running in User Mode. The USMAT contains pointers to the memory areas that the task may need to access while it is running in MCP Mode.

One Environment Table (ET) is constructed for each task on the system. The ET defines the addressing environment for each task. It accomplishes this by defining Memory Area Tables (and thus the functions or environments) that the task can use. Various MCP interfaces exist to change the addressing environment.

There is one Reinstate List for the entire system. There is one entry in the list for each task. The entry for each task contains dispatching information, state information, and the task's ET address.

THE MEMORY MANAGER

It is the job of the Memory Manager to perform all functions related to the allocation and management of memory. Several tables, described below, are maintained by this module:

- the Environment Tables,
- the Memory Area Tables,
- the Data Base Program Table, and
- the Pool Manager Table.

The Environment Tables and Memory Area Tables have been mentioned and will be discussed in detail later in this section.

The Data Base Program Table is used by all Data Base Program Tasks.

The Pool Manager Table is maintained in order to keep track of all used and unused Memory Area Tables and Environment Tables.

The Memory Manager's exclusive table memory area contains:

- the Memory Area Status Table,
- the Available List, and
- the Memory Table.

The Memory Area Status Table (MAST) contains an entry for each memory area on the system. Each MAST entry contains information on the size of the memory area, and the Task Number, Environment Number, and Memory Area Number it currently is assigned. This table is owned by the Memory Manager and shared with the hardware.

All available Memory Blocks are contained in a linked list called the Available List. Available Memory Blocks are the constituent elements for building Memory Areas.

When there is not sufficient room in main memory to continue to contain a Memory Area at a given instant, it can be written out to disk. The Memory Area's disk address and other information is kept in the The Memory Table entry for that Memory Area.

MEMORY AREA TABLE (MAT)

A MAT defines the addressing area for a particular task at an instant in time. Each entry in the table resolves to a memory area's Base/Limit pair, either directly or indirectly through the use of Copy Descriptors.

An entry consists of two fields: the Flag field and the Base/Limit field. The flag field value specifies the format of the Base/Limit field. The Base/Limit pair defines the bounds of the memory area.

MATs can contain from 1 to 100 entries. Upon reinstatement of a task, the first eight entries are loaded into the processor's memory area Base/Limit registers.

MEMORY AREA 0 AND MEMORY AREA 1

Memory Area 0 is always the local Data Area and Memory Area 1 is always the Code Area. When the MAT is loaded, the user can access up to one million digits of code (in Memory Area 1) and up to one million of the possible seven million digits of data (in Memory Area 0).

For pre-Omega code files, the Code Memory Area and the Data Memory Area are the same.

EXECUTABLE AND NON-EXECUTABLE MATs

A MAT can be either executable or non-executable. Each task's Environment Table entries contain the address of one of the task's MATs and the number of entries in that table. While a task is executing, the MAT that is loaded into the hardware defines the task's local addressing environment, and it is defined by an executable MAT. An executable MAT's Base/Limit entry #1 references a code memory area, and instructions are fetched from this memory area.

A non-executable MAT does not contain a reference to a code memory area in Base/Limit pair #1. Instead, all entries are used for storing memory area descriptors. These descriptors can be Base/Limit pairs, or they can be Copy Descriptors that will ultimately resolve to Base/Limit pairs.

USER SERVICES MAT (USMAT)

Every task has a User Services MAT (USMAT). This non-executable MAT describes MCP memory areas containing information about that task. In order to protect these memory areas from user access, the USMAT is located by the MAT address field in the User Environment Table entry #0 for that task. Attempts to access any of the memory areas through an instruction and an environment number of zero are legal only in Privileged Mode.

USER EXECUTE MAT (UEMAT)

The addressing environment for the task's user code is stored in a table called the User Execute MAT. The Environment Table for each task contains pointers to both the USMAT and the UEMAT.

MEMORY AREA TABLE ENTRY FIELDS

A MAT entry's type is determined by the first hexadecimal digit. All of the legal types are listed in Table 4-1.

Table 4-1. Memory Area Table Field Entries

FIRST HEXADECIMAL DIGIT	TYPE
0 - 9	Original
"A"	Invalid
"B"	Unused
"C"	Task-Dependent Copy
"D"	Invalid
"E"	Task-Independent Copy
"F"	Faulted

Present Original Entry

The Present Original (0-9 flag value) indicates that the memory area's Base/Limit pair is contained in this entry's Base/Limit field. This Base/Limit pair will be loaded into the processor's Base/Limit registers.

Unused Original Entry

If referenced, a MAT entry with the Unused Original (B flag value) will generate a Base/Limit error. The Unused Original designation indicates an entry that has not yet been filled with any of this task's memory areas.

Task-Dependent Copy Entry

The Copy (C flag value) indicates that the Present Original Base/Limit pair is contained in another MAT entry. The Base/Limit field contains the MAT number and MAT entry number of either another Copy entry, or of a Faulted entry.

A C-type Copy Descriptor has the following format:

C 0 <6-digit Environment Number> <2-digit Memory Area Number>

If the first digit of the Environment Number is a "D", then the next five digits are an index into the MCP ET. If the first digit is a digit "0 - 9", then all six digits are an index into this task's Environment Table. (The MCP ET is the Environment Table for Task #0.)

C-type Copies are evaluated every time a MAT loading operation is performed. If the Environment Number has the value zero or one, indicating a user Environment Number, then it is evaluated in the context of this particular task.

Because the C-type Copy Descriptor is always evaluated with reference to the task evaluating it, it is said to be "task-dependent".

For example, an entry value of C0<000000><00> is evaluated as a Copy of Environment 0, Memory Area Number 0 of the task that loads it. In this case, it will point to the MCP Data Area of the running task.

Task-Independent Copy Entry

The E-type Copy Descriptor (E flag value) has the following format:

E <9-digit absolute machine-dependent address of the Original>.

The V5 Series fully supports 9-digit absolute addresses. In the case of the V3 Series, which supports only 8-digit addresses, the high-order digit of the 9-digit address is set to "0".

E-type Copy Descriptors are created from a source of:

<Environment Number><Memory Area Number><Task Number>.

Because this information is taken at the time the Copy Descriptor is created, rather than at the time it is evaluated, the E-type Copy is termed "task-independent".

E-type Copy Descriptors can be built by the Memory Manager and by the privileged Alter Table Entry (ATE) instruction. (See Appendix B, ATE instruction.)

The E-type Copy is useful when it is necessary to have access to the addressing environment of a task for whom the current task is not running. For example, the I/O Complete Independent Runner (IOT I.R.) may need to post a Buffer Status Word (BSW) to a user program. IOT I.R. is not called by the user, rather, it is called by the Kernel in response to the asynchronous external request brought about by the I/O completion. Through the Queue Element, the IOT I.R. is able to identify the program and obtain the address within the program where the BSW is to be posted. IOT I.R. picks out the Task Number, Environment Number, and Memory Area Number, puts them in a template, and then executes the ATE instruction. Thus, the Copy will point to the Original for the task in which IOT I.R. needs to post the BSW.

Faulted Entry

The Base/Limit field corresponding to a Faulted entry (F flag value) contains a Fault Index and a Memory Area Status Table Number. (The Fault Index is for software use only.)

Refer to Section 4, Memory Area Address Resolution, for a detailed discussion of how Memory Area Table entries are resolved.

ENVIRONMENT TABLE (ET)

One Environment Table is built for each task on the system, including the MCP functions. An ET can contain from 1 to 1,000,000 entries. Each ET entry contains an absolute address pointer to a MAT. The ETs are system arrays built and maintained by the MCP to inform the processor of a task's legal addressing environments.

Since a task can execute instructions originating from user code as well as instructions from MCP code modules, a task needs the capability of accessing environments in the MCP ET and the task's User ET. Each MCP or User ET entry points to a MAT that describes the local addressing environment for that task at its current point of execution.

The MCP ET contains an absolute address for each MCP code module MAT. Each MCP MAT defines the addressing environment for that code module once it has been invoked by a user task. The MCP ET is located by the ET address field in the Reinstate List entry for task #0. This Environment Table is shared by all tasks.

Every task has its own User ET. It is located by the ET address field in the Reinstate List entry for that task.

A user task ET contains three entries, each of which contains an absolute address pointer to a MAT. One entry points to the User Execute MAT (UEMAT). A second entry points to the User Services MAT (USMAT). A third entry is available for the Alter Table Entry operation. (See Appendix B, ATE instruction.)

Refer to Figure 4-1 for a schematic diagram depicting the relationship of the ET to the MATs.

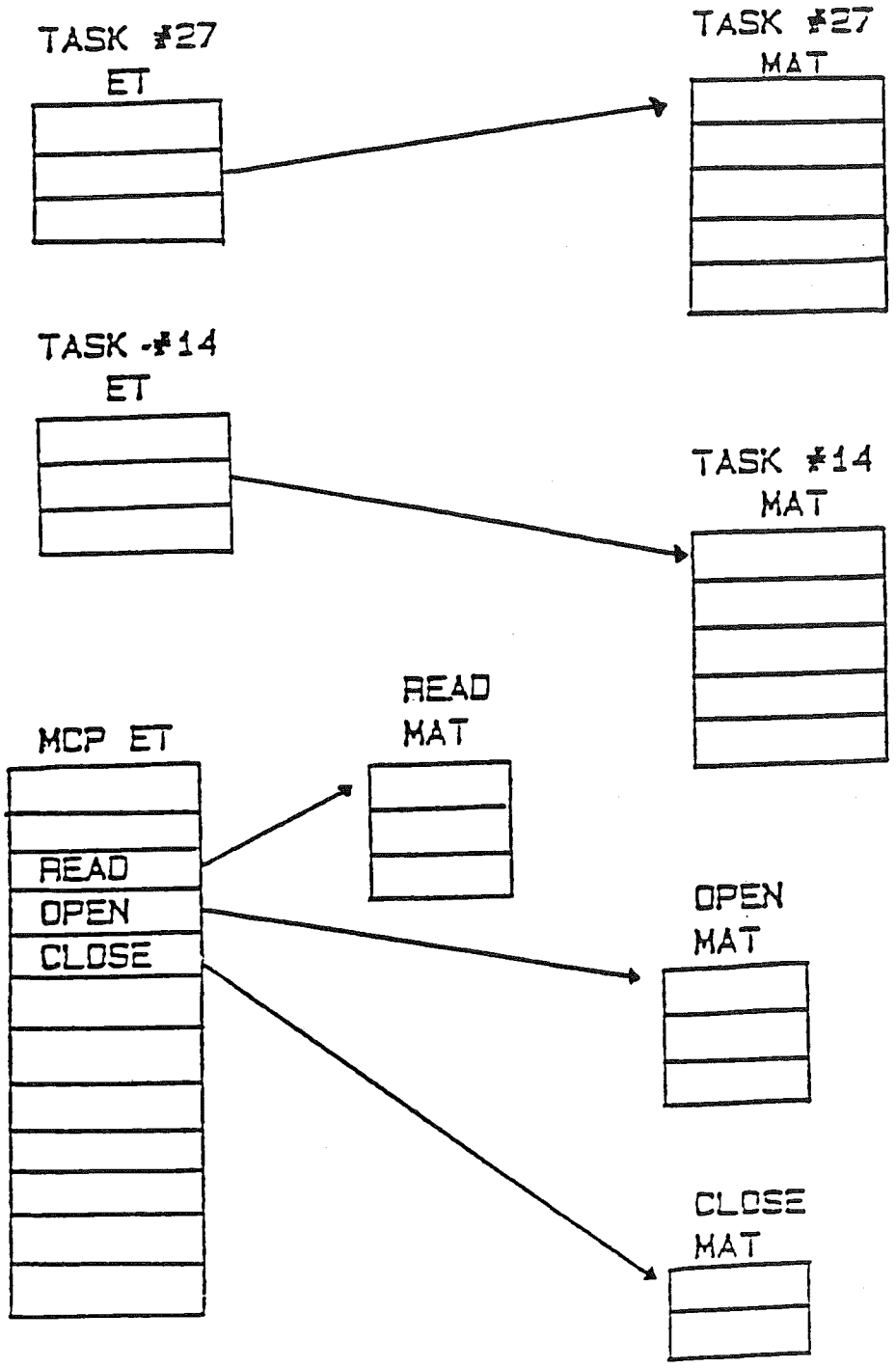

Figure 4-1. ET - MAT Relationship

REINSTATE LIST

The Reinstate List is a system array set up by the MCP in order to control task switching for the processor. Every task on the system has an entry in this table. An entry contains a pointer to the ET for that task, as well as other important information.

Two entries have special significance. The entry for task #0 is not assigned a task. This entry's ET address field is reserved for the address of the MCP ET. Similarly, the entry for task #1 is reserved for the MCP Kernel code ET address.

LOCATING A MEMORY AREA TABLE ENTRY

A MAT entry is specified by a Task Number (TN), an Environment Number (EN), and a Memory Area Number (MAN). To locate a MAT or an entry within a MAT, one must traverse the address links through the Reinstate List, the MCP Environment Table or the task's User Environment Table, and Memory Area Tables.

The Task Number represents an array subscript into the Reinstate List of 0000 to 9999. (The actual number of possible tasks can be limited by memory constraints.) The address of the Reinstate List is provided by software by the Write Hardware Register (WHR) instruction. The processor maintains an internal pointer to the Reinstate List entry for the current task, but must recalculate any references to Reinstate List entries for other tasks. (See Appendix A, the BF=00 variant of the WHR instruction.)

If the first digit of the EN is equal to a "D", then the five least significant digits represent an array subscript into the MCP ET of 00000 to 99999. If the first digit is equal to the values of "0" through "9", this six-digit number represents an array subscript into a User ET of 000000 to 999999. (Again, the actual number of environments for a task can be limited by memory constraints.)

An Address Error Fault will occur if the value in the first digit is any other value, or if the array subscript portion of the EN is larger than the Number of Entries field of the selected ET.

The Environment Table entry that has been successfully located will contain the address of the desired Memory Area Table in its Memory Area Table Address field.

An Address Error Fault will occur if the Memory Area Number parameter is greater than the MAT's Number of Entries field, or if any digits of the Memory Area Number contain undigits.

MEMORY AREA ADDRESS RESOLUTION

As discussed, a MAT entry can contain the following flag value types: a Present Original type (0-9), an Unused type (B), Task-Dependent Copy Descriptor type (C), Task-Independent Copy Descriptor type (E), or the Memory Area Fault (or Absent Original) type (F).

Entries of Copy Descriptor type provide levels of indirect addressing for MAT entries. Only one Original entry exists for each Memory Area, and a single Original entry can have several Copy Descriptor type entries pointing to it.

If the MAT entry being resolved has a flag value of Copy Descriptor (C), then the information in this entry must be used to locate another MAT entry, which in turn must be resolved. If a chain of Copy Descriptors exists, then the resolution process will continue until a non-Copy Descriptor flag is found at the end of the chain.

If the final MAT entry's flag is of Original type, then the Base/Limit pair contained in this entry is loaded directly into the processor's Base/Limit registers.

If the final MAT entry's flag is of Memory Area Fault type (F), then the processor will cause a Memory Area Fault; the Fault Handler is then invoked. The Fault Handler in turn calls the Memory Manager that attempts to make sufficient space available to roll-in the faulted area. If this operation is successful, the Memory Manager obtains that Memory Area's disk address from the Memory Table. It then initiates an I/O to read the disk contents into the allocated memory and then overwrites the MAT entry with the value "0" and the Base/Limit pair. The processor will then reload this MAT and will, upon finding the newly created Original entry, load the corresponding Base/Limit pair into its Base/Limit registers. (See Section 10, Soft Memory Area Faults and Section 10, Hard Memory Area Faults.)

A flag value of "B" indicates that no memory is assigned to this Base/Limit pair. The Base and Limit register values are set equal to each other (zeros) for Unused entries. The processor will still load the Base/Limit pair, and thus any attempt to access memory through an Unused entry will cause an Address Error fault.

MEMORY AREAS 8 THROUGH 99

Although only the first eight MAT entries are loaded by the processor, entries 8 - 99 prove useful as well. They contain Base/Limit pairs that point to other memory areas that the task may need to access during execution.

For example, a Copy Descriptor in Memory Areas 2 - 7 can be changed to point to any of entries 8 - 99, which in turn may point to a buffer. Assume that entry 52 points to this buffer. It is a simple matter for the processor to access the buffer once one of the entry 2 - 7 Copy Descriptors is made to point to entry 52.

The privileged instruction used to alter a MAT entry is the Alter Table Entry instruction. (See Appendix B, ATE instruction.)

Refer to the following page for a schematic diagram depicting the MAT - Memory Area relationship.

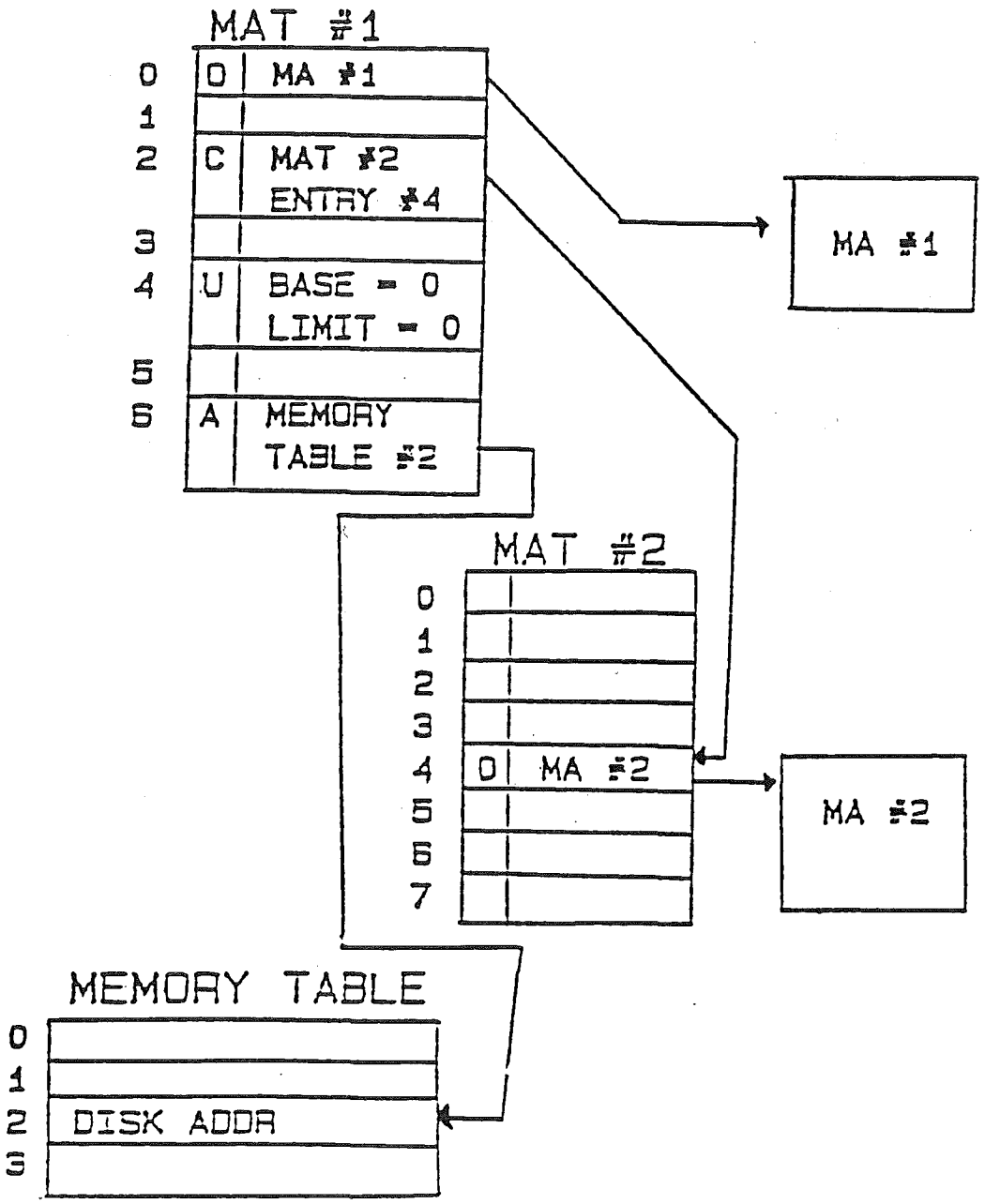

Figure 4-2. MAT - Memory Area Relationship

MEMORY AREA PUSHING

The Memory Manager can physically move memory areas about in real memory if there is not enough contiguous space available to satisfy a new request for space allocation, but there is enough total available space. This process is known as pushing. The objective is to reduce the amount of memory fragmentation and increase the size of the contiguous available memory areas.

Because of the existence of only one Original for a memory area, the memory area can be pushed transparently to the task, provided that certain I/O related constraints are followed. (See Appendix B, the Convert I/O (CIO) and I/O Complete (IOC) instructions.)

Pushing generally has a lower overhead than Rollin/Rollout.

MEMORY AREA ROLLIN/ROLLOUT

Rollout occurs when insufficient memory is found in the Available List for a task requesting it. For example, Rollout occurs when Beginning of Job (BOJ) cannot be granted sufficient memory for the code file.

If the task's memory priority is high enough, it will cause memory areas belonging to another task to be rolled out. The Original of any memory area being rolled out will be marked as a Faulted entry.

When the task that owns the rolled-out memory area(s) is reinstated, it will incur a Memory Area Fault after it attempts to load the faulted entry. The Fault Handler is called, and it in turn calls the Memory Manager, and passes the Memory Area Status Table Index to the Memory Manager. In effect, the task requests that the missing memory area be rolled in. In the process of rolling in the missing (faulted) memory area, it may be necessary for the rollout routine to be called.

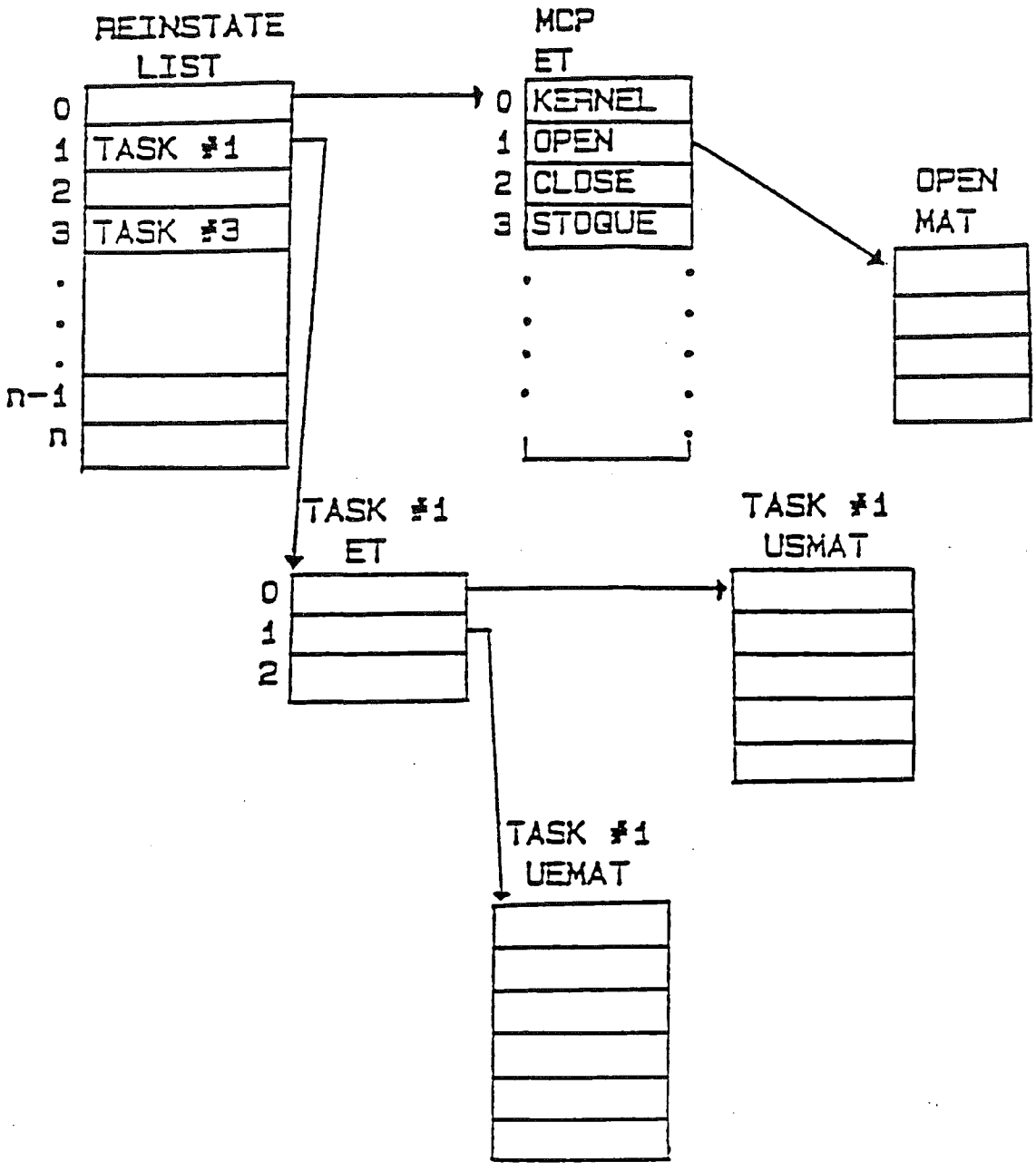

Figure 4-3. MAT, ET, and RL in Memory

Table 4-2. Memory Area Status Table (MAST) Entry Format

DIGIT	BIT	PURPOSE
00-01		Hardware Lock (for use by the processor)
02	3	Memory Area Present
02	2	To Be Rolled Out
02	1	Reserved
02	0	I/O Inhibited Memory Area
03-05		Software Usage
06-09		Number of I/Os in Process
10-13		Task Number of Owner
14-19		Environment Number of Original
20-21		Memory Area Number of Original
22-25		Memory Area Size
26-39		Available

Table 4-3. Reinstate List (RL) Entry Format

DIGIT	PURPOSE
000-007	Link to Next Reinstate List Entry
008	Soft Fault Pending Flag
009	I/O Flags
010-015	Number of Entries in Environment Table
016-018	Environment Table Address Expansion Area
019-027	Environment Table Address
028-037	Failed Hardware Call R/D Area
038-039	Task Processor Priority
040-043	Task Number Owning
044-047	Next Task on List
048-049	State Indicator
050-053	MCP Canonical Lock Number
054-057	User Canonical Lock Number
058-061	Operating Claim
062-070	Next Scheduled Run Time
071-076	Task Wait Time
077-082	New Time Slice
083-090	Direct Time Accumulated
091-092	Mode Indicator Save Area
093-101	Software Usage
102-105	Task Number
106-113	Time Slice Remaining
114-199	Interrupt Frame:
114-141	Accumulator
142-149	Measurement Register
150-151	Interrupt Mask
152-183	Mobile Index Registers
184-185	Mode Indicators
186-187	COM and OVF Flags
188-193	Active Environment Number
194-199	Instruction Address

SECTION 5
TASK EXECUTION

GENERAL

Most MCP functions run as subroutines of the user task. That is, when the MCP has the processor, the MCP is running for the calling user task. The addressing environment for the MCP is defined by the MCP ET and the ET of the task for which MCP is running. The following two sections show the relationship between the user task and the MCP. In the example, a user task (Task #2) first executes user code and then makes a call to the MCP for a Read function.

THE USER ENVIRONMENT WHILE USER CODE IS EXECUTING

Figure 5-1 depicts the addressing environment and what the memory management structures look like while user code is executing in User Mode.

The processor is given the address of the first entry of the Reinstate List (through the Write Hardware Register instruction, WHR) at Coldstart.

In this illustration, User Task #2 is to be dispatched. The RL entry for Task #2 contains the absolute address of the task's Environment Table.

Entry #0 of Task #2's ET points to the User Services Memory Area Table (USMAT). The USMAT is not used in User Mode, except in order to resolve Copies.

Entry #1 of the task's ET points to the User Execute MAT (UEMAT). In User Mode, the processor loads the UEMAT. Entries #0 and #1 are Copies and resolve to Entry #1 of the USMAT (an Original). This Original entry points to the User Data and Code Memory Area.

Entries #2 - #7 of the UEMAT are Unused in this example. The first release of MCP/VS allows user programs to access memory areas 2 - 6. (For example, new COBOL compiler syntax for 2.0 is implemented to allow this.)

Since a task operating in User Mode can access only the memory areas defined by the MAT it is executed in, it cannot access or corrupt any code or data structures belonging to the MCP or another task.

This seemingly elaborate protocol is necessary in order to assure that old code files, ignorant of context addressing, can run compatibly. Context addressing sets forth specific rules about which memory areas operands are resolved relative to, depending on whether or not there is indexing on the operands. (See Section 3, Address Resolution.)

Because Task #2 is running in User Mode, it has no ties to any of the MCP data structures or code. Thus, this task cannot access or corrupt any MCP data structures as long as it is executing user code.

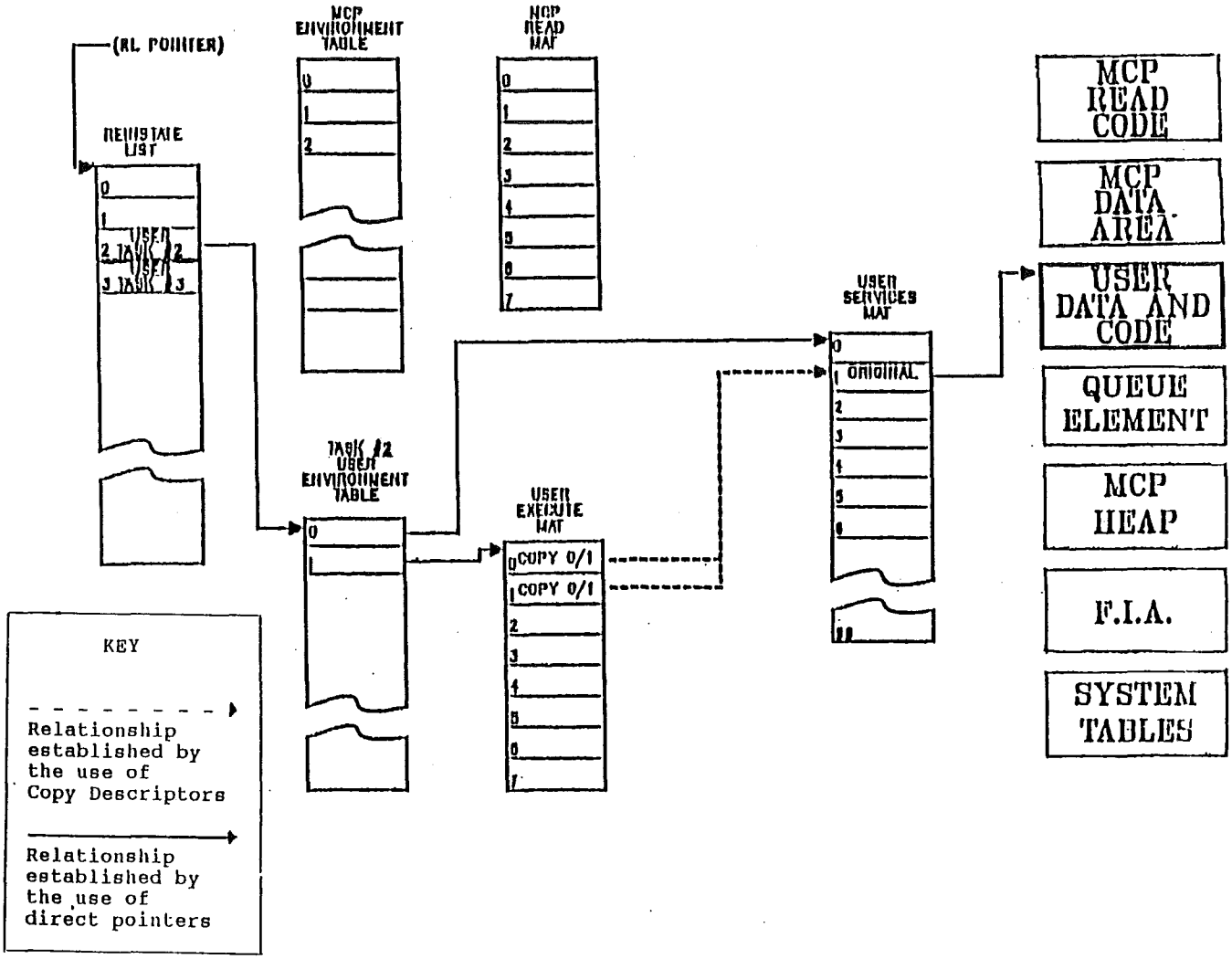

Figure 5-1. User Environment While User Code is Executing

THE USER ENVIRONMENT WHILE MCP CODE IS EXECUTING

Figure 5-2 depicts the user environment while MCP code is executing for a task. The MCP Read function is to be performed for Task #2.

The RL entry for Task #2 points to the task's ET. Entry #0 of Task #2's ET points to the USMAT.

In MCP Mode, the USMAT entries (Originals) point to many more data structures than is the case in User Mode. In User Mode, only the User Data and User Code are essential to the execution of user code.

The USMAT points to the data structures that the MCP may need to access during execution of the MCP function.

Entry #0 of the USMAT points to the MCP Area. One MCP Data Area is allocated to each user task. It contains the stack that the task will use while running in MCP Mode. It also contains the IX1, IX2, IX3 register contents, the stack pointer, and other user-specific information. (See Table 5-1, Reserved Memory Relative to the MCP Data Area.)

Entry #1 points to the User Data and Code.

Entry #4 points to the I/O Queue Elements that are maintained for opened files. There is one Queue Element assigned per buffer per file that the user has opened. The Queue Element is used in initiating I/Os and storing Result Descriptors from the I/O Subsystem.

Entry #5 points to the MCP Heap, which contains buffers for pseudo devices (such as printer backups, pseudo card readers, and pseudo card punches) and address blocks for disk and pack files.

Entry #6 points to the File Information Area (FIA). There is one file information structure in the FIA for each file that the user has open. This file information structure contains IOATs for disk and pack files, external FIBs, and disk and pack file headers if any disk or pack files were opened.

Recall that the MCP Read function is to be performed. Entry #0 of the RL always points to the MCP ET. Each entry in the MCP ET points to a MAT for an MCP function.

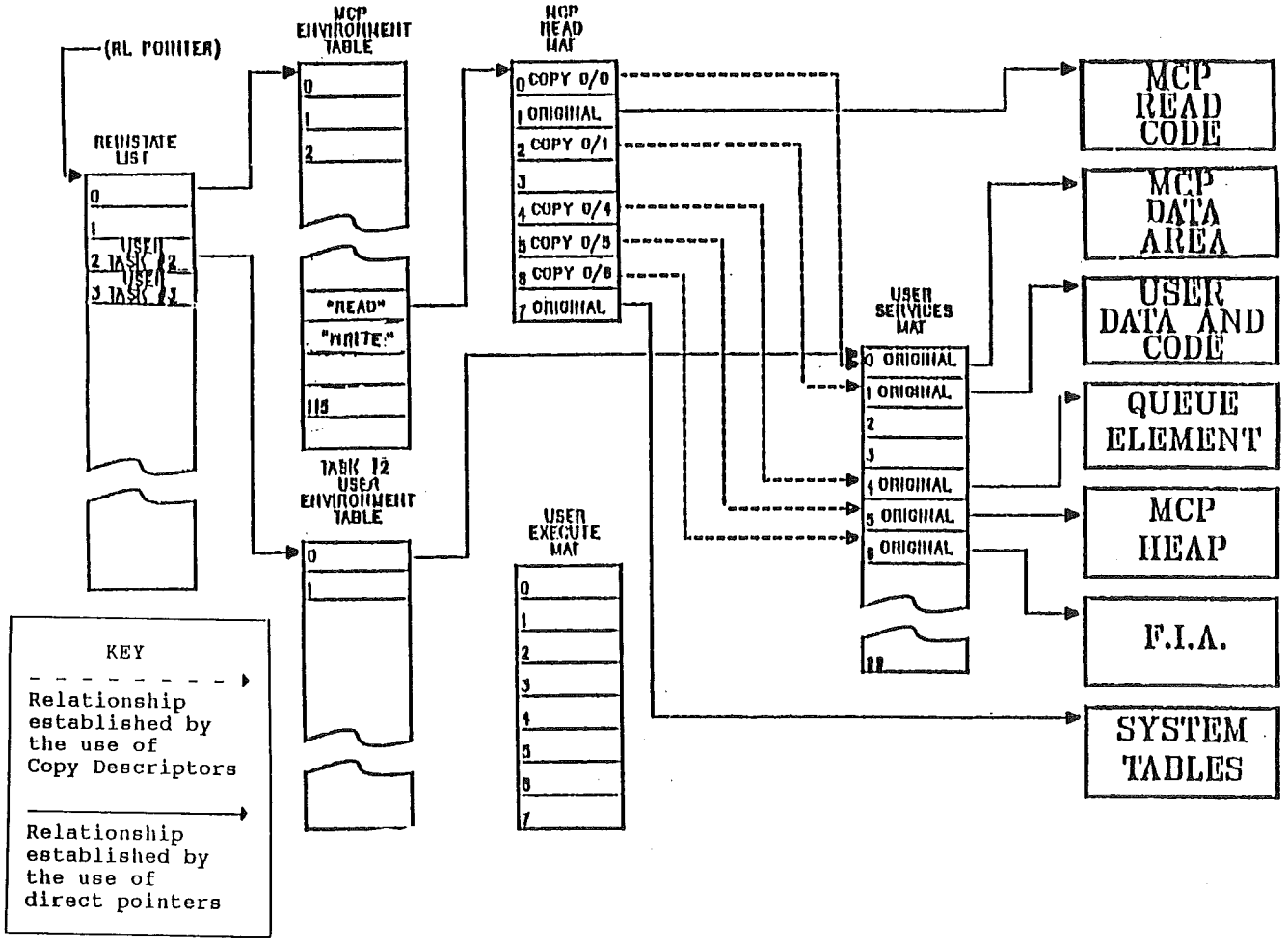

Figure 5-2. User Environment While MCP Code is Executing

Entry #0 of the MCP Read MAT contains a Copy to the task's ET Entry #0, Memory Area #0. That is, Entry #0 contains a Copy Descriptor that points to a MAT entry, the USMAT Original. The Original points to the MCP Data Area.

Entry #1 contains an Original entry pointing to the MCP Read code.

Entry #2 is a Copy of ET Entry #0, Memory Area 1, which resolves to User Data and Code. The Read function must have access to the user data structures and code (through the USMAT) because it will need the FIBs, buffers, and buffer status words in order to execute.

Entries #4-#6 are Copies of the Queue Element area, The MCP Heap, and the FIA, respectively.

Entry #7 is an Original that points to the MCP System Tables.

While the Read function is executing, it has addressability to memory areas containing its own code, MCP data, user data and code, the I/O Queue Elements, the MCP Heap, and FIA area for this particular task. A separate set of those memory areas is kept for each task on the system.

While a task is executing MCP code, the operating system has access only to those memory areas belonging to that task. Copy Descriptors are the features that allow for this limited access of memory areas.

A C-type (Task-Dependent) Copy Descriptor can resolve only to the ET and memory areas of the current task. Memory areas belonging to other tasks are completely inaccessible to the current task, because it is physically impossible to give the processor another task number in the Copy Descriptor field format. All C-type Copy Descriptor entries are relative to the current task.

Table 5-1. Reserved Memory Relative to the MCP Data Area

RELATIVE MEMORY ADDRESS	PURPOSE
00 - 39	Indirect Field Length
08 - 15	Index Register 1 (IX1)
16 - 23	Index Register 2 (IX2)
24 - 31	Index Register 3 (IX3)
38 - 39	SCAN Result Storage
40 - 45	Stack Pointer
46 - 47	Breakpoint Bit Pattern
48 - 49	Edit Table Entry 0
50 - 51	Edit Table Entry 1
52 - 53	Edit Table Entry 2
54 - 55	Edit Table Entry 3
56 - 57	Edit Table Entry 4
58 - 59	Edit Table Entry 5
60 - 61	Edit Table Entry 6
62 - 63	Edit Table Entry 7
64 - 65	Trap Enable (FF)
66 - 71	Trap Address
72 - 81	Result Descriptor Storage Area
82 - 85	Task Number
86	Reserved
87 - 92	Hyper Call Function Table Pointer
93	Reserved
94 - 99	Hyper Call Function Table Limit
100 - 113	Reserved
114 - 1014	BCT Interface Area

SECTION 6
CODE AND DATA PROTECTION

CODE AND DATA INTEGRITY

In an environment where multiple users have access to shared data structures, it is essential to protect those structures as well as code and data belonging to the individual processes.

A task's data and code are afforded protection from other tasks' attempts to overwrite its memory by virtue of two design features. The first feature is the base and limit checking that the hardware performs. The second feature has to do with the way that the ETs and MATs are linked, depending on whether user code or MCP code is executing.

BASE AND LIMIT VALUES

Memory area protection is provided by comparisons of the base and limit values to the resolved addresses of the requested memory. This is to ensure that the value of a requested address is less than the limit value, but greater than or equal to the base value. If the address of a requested memory area lies outside of the specified memory area, then an Address Error fault is caused and the instruction is terminated.

EXECUTION ENVIRONMENT

The addressing environment for the MCP is defined by the MCP ET and the user task's ET. To the user, the MCP ET appears logically to be an extension of the user's own task ET. Physically, however, there is one MCP ET.

Due to the fact that the MCP in fact runs as a subroutine of the user task, if a task harms the MCP, no one else's operating system is corrupted except that user's. That is, a user failure is contained so that it harms only the offending user, provided that no system-wide data item has been corrupted.

While a task is running in User Mode, it has no ties to the MCP data structures or code. Thus, a user cannot access or corrupt any MCP data structures as long as he is executing user code. In effect, a designed partitioning of the ET and MAT structures exists according to whether the code that is running belongs to the user or the MCP. Refer to Figures 5-1 and 5-2 for an example of the environment during user and MCP code execution.

MULTI-THREADING

Many users may be in the operating system, simultaneously performing the same function. In other words, an MCP task need not run to completion in order for other users to be allowed into the operating system.

The fact that multiple users may be in the operating system and simultaneously accessing the same MCP function code means that the MCP is a multi-threaded operating system. A versatile locking mechanism is the cornerstone for the protection of this multi-threaded operating system.

LOCKS

Locks allow multiple users to share data structures. Other users must be locked out of a shared data structure until a single task has finished updating the structure's contents. The locking mechanism preserves the data integrity of a shared data structure. There are two types of locks: Canonical Locks and Events.

CANONICAL LOCKS

A Canonical Lock has an associated Canonical Lock Number, or Lock Level. The Lock Level dictates the order in which locks are obtained and released, and is essential for preventing deadlocks.

If a user owns a lock on a structure but does not have the processor, there is a possibility that other tasks will want to access that structure. If other tasks attempt to acquire the lock and have to contend for that lock, the task owning that lock will have its Operating Claim raised by a certain value.

The Operating Claim is a measure of the task's right to run. It is similar, in effect, to a task's processor priority, but it is separate from the task's priority. The value that the lock-owning task's Operating Claim is increased by is that of the contending task's Operating Claim.

The task that owns the lock will have its Operating Claim raised as long as other tasks are in contention for that lock. At some point, the task's Operating Claim may become so high that the operating system will be forced to reinstate the task immediately in order for the task to finish with the structure and then release the lock. Therefore, no task owning a lock for which there is contention can tie up the system. This is true for tasks of high as well as low priority.

EVENTS

An Event signals that a certain event has occurred on the system. Events are used to synchronize two independent streams of code.

A task that is using a certain resource can signal an Event. Resources in use by a single task are signalled as being off-limits to contending tasks until the first task signals that the event is over. Those tasks waiting on that event will then be dispatched.

SECTION 7
THE KERNEL

GENERAL

The Kernel is a special operating system task, which:

- handles all processor interrupts,
- determines the next task to be dispatched to the processor, and
- contains several system maintenance routines that require an implicit system-wide lock for their safe use.

The Kernel is the MCP's innermost layer and is composed of two portions: the Interrupt Handler and the Dispatcher. Both of these Kernel functions maintain data structures for their own use, and the use of other operating system functions, such as time accounting.

The Kernel must be capable of functioning as a mini-operating system by performing crucial functions for the rest of the operating system. As such, the Kernel runs as a separate task, and has its own Reinstate List entry, state and index registers, Interrupt Mask, and MCP Data Area, the Interrupt Data Page. It also has a privileged relationship with the processor hardware/firmware. During the course of interrupts, state and other information are stored in these special registers.

The Kernel can be invoked only by an interrupt such as that caused by the I/O or Timer Interrupt, or the Interrupt instruction (INT, OP=90). As mentioned, the Kernel's responsibility is to service interrupts and to determine the next task to be run. The Branch Reinstate Virtual instruction (BRV, OP=93) informs the processor which task to reinstate. Thus, the Kernel is the only task that will execute the BRV instruction (except for the System Loader program that is responsible for the initial stages of system initialization).

THE KERNEL INTERFACE

There are numerous interfaces to the Kernel that exist in the form of requests. Regardless of the fact that there are numerous Kernel functions, the Kernel is always entered as a result of an interrupt. The interrupt can be forced by:

- the hardware (I/O and Timer Interrupts),
- the firmware (Non-Maskable Instruction Interrupts such as failed Lock), or
- the software (Operating System Kernel Requests).

In the case a software interrupt, a Kernel Request is made asking the Kernel to perform a specific function. Usually the function involves the movement of Reinstate List entries from one list to another, or modification of the task state. In any case, the funtion's unique identification is represented as a parameter to the interrupt call. (See Section 7, Kernel Requests.)

KERNEL DATA STRUCTURES

In addition to its two logical parts, the Interrupt Handler and the Dispatcher, the Kernel contains the support routines for managing the Kernel's internal data structures. This will usually entail operating on one such structure, the Ready List.

The Kernel's Interrupt Handler determines which Kernel routine is being requested and performs that function.

The Kernel's Dispatcher always reinstates the task at the head of the Ready List.

What follows is a discussion of the Kernel's data structures. They are special purpose lists of specially ordered tasks. Tasks not contained in those lists are discussed last.

Ready List

The key data structure in the Kernel is the Ready List. This is the list from which the next executable task is chosen once the processor is free to take on the next task.

Entries in the list are ordered according to their Operating Claim. (See Section 11, Operating Claim.) Tasks are relinked into the list in a round-robin fashion within a given Operating Claim.

The Ready List is maintained by linking together the Reinstate List entries of the Runnable Tasks in the order just discussed. The task whose entry is at the head of the Ready List is reinstated next.

This list can never be empty. In the event that no other tasks are on the system, the Idle task will be dispatchable.

Doze List

The second key Kernel data structure lists all the dozing tasks. It is called the Doze List. It is maintained by linking together all of the Reinstate List entries of the dozing tasks.

The Doze List is ordered according to the wake-up time of each task. The wake-up time is stored by the Doze routine in the Next Scheduled Run Time field of the task's Reinstate List entry before the Doze Request is made of the Kernel.

The Doze and Ready Lists are mutually exclusive. That is, a task can be on only one of these lists at a time.

Lock/Event Wait Lists

There are multiple Lock/Event Wait Lists in the Kernel. There is exactly one Wait List for each Lock or Event on the system for which there are currently one or more tasks waiting.

The head of a Lock Wait List is indicated by the Task Number in the Lock Waiter Link field of the Lock Structure of that Lock. The other tasks in the Lock Wait List are linked together using the Next Task on List field in the Reinstate List entry.

Each Lock Wait List thus represents the set of tasks that will be awakened by the Kernel when the firmware performs an Unlock for a Lock or a Cause for an Event.

Note that the Lock Wait Lists are mutually exclusive. No task can be on more than one Lock Wait List because a task can be waiting on only one lock at a time. Furthermore, if the task is in a Lock Wait List, it cannot be in either the Doze List or the Ready List.

It is the processor firmware (the Unconditional Lock and Event Wait Instructions) that enters tasks into a Lock Wait List. It is the Unlock and Cause that, when there are tasks waiting, cause a non-maskable interrupt to invoke the Kernel so that the waiting tasks can be relinked into the Ready List. The processor firmware performs the necessary housekeeping for the Lock/Event structure. (See Section 6, Locks and Events.)

Terminating List

A task that is terminating is linked onto this FIFO queue. The pointers to the beginning and the end of the list are maintained in the Interrupt Data Page.

A task is linked onto the end of the Terminating List when a Terminate Task request (a Kernel Request) is performed. The Task Terminate Independent Runner will later deal with this task and all other tasks in the Terminate List. (See Section 7, Get Next Terminated Task Request.)

Failed Task List

This list is similar to the Terminate List, except that it contains the list of tasks that have come to the Kernel's attention because of a Failed Hardware Call. (See Section 7, Get Next Failed Task Request.)

Tasks Not on the Kernel Lists

The only Reinstate List entries not on any of the above lists are:

- those not associated with a task. For example, the available Reinstate List entries are kept in the Memory Manager's Available List.
- the Reinstate List entries associated with Tasks #0 and #1 (the MCP Dummy Reinstate List entry and the Kernel, respectively). These tasks cannot be dispatched.
- any special Independent Runners that have not been dispatched directly from the Kernel and are not currently running. Among these are the Real-Time and Normal I/O Complete I.R.s, Task Ter-
  minate I.R., and System Fail I.R.

KERNEL REQUEST CATEGORIES

The Kernel Requests fall into three groups:

- Parallel,

- Firmware/Software, and

- Operating System Kernel Requests.

The three groups and their constituent members are listed in Tables 7-1 through 7-3.

PARALLEL KERNEL REQUESTS

Requests in this category can occur in parallel with each other. That is, they can occur alone or concurrently with any of the other requests. All of the Parallel Kernel Requests are listed in Table 7-1, and the individual parallel requests are described below.

Normal I/O Complete

This request indicates successful I/O completion on a non-real-time device. It causes the Normal I/O Complete (IOT) I.R. to be executed by linking the Normal IOT I.R.'s Reinstate List entry to the head of the Ready List. The Task Number of the Normal IOT I.R. is passed to the Kernel through an MCP Kernel Request, Pass Independent Runner's Task Number.

Error I/O Complete

This request is similar to the Normal I/O Complete request, except that it indicates that an error has been detected by the I/O Subsystem on a non-Real-Time I/O Complete.

Real-Time I/O Complete

This request indicates I/O completion on a real-time device. In a manner identical to that of the Normal I/O Complete Request, the Real-Time I/O Complete Request requests the Kernel to link the Real-Time I/O Complete I.R. to the head of the Ready List.

Timer Interrupt

This indicates to the Kernel that the Time Slice for the current task has been exceeded. The Kernel is requested to give a new Time Slice to the task, perform time accounting functions for the task, and determine if there is another task of equal Operating Claim that should be dispatched. (See Section 11, Operating Claim.)

System Overtemperature

The Kernel is requested to stop activity on the processor due to an overheating condition.

Non-Maskable (Firmware/Software)

This indicates that a Firmware or Software Kernel Request has been made. These requests will be discussed in a following section.

IDENTIFYING PARALLEL KERNEL REQUESTS

Parallel requests are indicated by a bit mask known as the Interrupts Occurred Byte, or Interrupt Descriptor, located at Kernel Data Area addresses 21-22. (Refer to Table 7-1, Parallel Requests and the Associated Interrupts Occurred Byte Codes and Table 7-4, Kernel Data Area). The bit mask stored here indicates each interrupt that was both pending and allowed by the Interrupt Mask. This byte's information structure, a bit mask, reflects the fact that any combination of interrupts can occur from this category. Of course, because non-maskable interrupts are always allowed there is no corresponding bit in the Interrupt Mask. (See Section 9, Interrupt Descriptor, for this mask's bit values and Section 9, Interrupt Mask.)

In the case of a Non-Maskable Interrupt, the code indicating the specific request is found in the Instruction Interrupt Cause Descriptor field of the Kernel Data Area (address 32-33). Those requests are discussed next.

Table 7-1. Parallel Kernel Requests and the Associated Interrupts Occurred Byte Codes

REQUEST	KERNEL DATA AREA ADDRESS	BIT
Non-Maskable Request (Firmware/Software)	21	4
System Overtemperature	21	2
Timer Interrupt	21	1
Real-Time I/O Complete	22	4
Error I/O Complete	22	2
Normal I/O Complete	22	1

FIRMWARE/SOFTWARE KERNEL REQUESTS

These requests cannot occur concurrently with each other, although they can occur simultaneously with Parallel Requests. This group includes all of the firmware-forced interrupts.

Failed Lock

This indicates that the hardware has attempted to perform an Unconditional Lock variant of the Lock instruction against a lock that another task already owns. The firmware requests the Kernel to suspend the current task until the lock is released and then links this task into the Lock List. (See Appendix B, LOK instruction.)

Failed Event

This request indicates that the firmware has attempted to perform a Wait variant of the Lock instruction against an Event that has not been caused. The Kernel is requested by the firmware to suspend the current task until the event on which this task is waiting is caused by another task. (See Appendix B, LOK instruction.)

Released Contended Lock

This request signals that the firmware just completed the execution of an Unlock variant of the Lock instruction against a Mutual Exclusion Lock that has other tasks suspended. The Kernel is requested to link the waiting tasks into the Ready List and to adjust the Operating Claim of the current task to reflect the fact that this task is no longer blocking the other tasks. (See Appendix B, LOK instruction.)

Released Contended Event

This request signals that the firmware completed the execution of the Cause variant of the Lock instruction against an event that has tasks suspended. These tasks are waiting for the event to be released. The Kernel is requested to link those suspended tasks into the Ready List. (See Appendix B, LOK instruction.)

Failed Hardware Call

The firmware just failed an attempt to execute a Hardware Call Procedure for the current task.

Interrupt Instruction (OP=90)

This indicates that an MCP Kernel Request has been made.

IDENTIFYING FIRMWARE/SOFTWARE REQUESTS

Any request in the Firmware/Software category will cause the Non-Maskable Interrupt bit in the Interrupt Descriptor to be set. The cause of the interrupt is found in the Interrupt Cause Descriptor byte, located at addresses 31-32 of the Kernel Data Area.

Since there can only be a single non-maskable interrupt at a time, the information in this byte is represented by a single value, rather than in a bit mask. (See Table 7-2, Firmware/ Software Requests and the Associated Interrupt Cause Descriptor Codes for this byte's values.)

In the case of the Interrupt instruction, the Interrupt Cause Descriptor byte has the value (06), and the MCP Kernel Request code is found in the MCP Kernel Request Code field of the Kernel Data Area (address 34-35). The MCP Kernel Requests are discussed next.

Table 7-2. Firmware/Software Requests and the Associated Interrupt Cause Descriptor Codes

REQUEST	INTERRUPT CAUSE DESCRIPTOR CODE
(Reserved)	00
Failed Lock	01
Failed Event	02
Released Contended Lock	03
Released Contended Event	04
Failed Hardware Call	05
Interrupt Instruction (OP=90)	06

MCP KERNEL REQUESTS

These requests can be forced from the software. Only one of these requests can occur at a time.

The MCP interface with the Kernel is made through the Interrupt instruction, along with one of the Kernel Request codes. The Kernel Requests are special requests made of the Kernel by the MCP. Usually the requests are to perform various task management functions by manipulating the Kernel's own data structures, such as the Ready and Doze Lists.

HOW MCP KERNEL REQUESTS ARE INVOKED

The MCP causes a Kernel Request by executing the Interrupt instruction (INT, OP=90), and including the Kernel Request Code among the instruction's parameters. (See Appendix B, Interrupt Instruction.)

Some Kernel Request Codes also require data to be passed along as parameters for the particular request being serviced. The INT instruction's AF field indicates this when it contains a non-zero value indicating the data length.

FUNCTIONAL DESCRIPTIONS OF THE MCP KERNEL REQUESTS

Doze Current Task

The Kernel is requested to suspend a task for a certain period of time. The task will be scheduled to run next at the time, in milliseconds since midnight, stored in the Next Scheduled Run field of the task's Reinstate List entry.

The current task must store the wake up time in the Next Scheduled Run Time field of its Reinstate List entry. The routine then removes the task from the Ready List and links it into the Doze List, based on its new Next Scheduled Run Time field value.

Awaken Dozing Task

The Kernel is requested to awaken the dozing task that is indicated in the data passed along with the Interrupt. If this task is found in the Doze List, it is delinked from this list and linked into the Ready List, based on its Operating Claim. If the task is not found in the Doze List, the request is ignored.

Adjust Dozing Period

The Kernel can change the wake up time of a task that is currently dozing. The Task Number for this task and its new wake up time must be passed along as data with the Interrupt.

The Kernel searches the Doze List for the indicated task. If the task is found, it is first delinked from the Doze List. The new wake up time is stored in the Next Scheduled Run Time field of its Reinstate List entry, and the task is then relinked back into the Doze List based on this new time value. If the task is not found in the Doze List, the request is ignored.

Terminate Current Task

The Kernel can be requested to remove the current task from the active mix and cause the Task Terminate Independent Runner to be executed.

This routine delinks the current task from the Ready List and sets its state to Waiting Terminate. If the Task Terminate Independent Runner is not running, it is linked into the Ready List. The Task Number of the task to be terminated is stored in system tables. If Task Terminate is already running, the terminating task is linked into the Terminate List.

The Task Terminate Independent Runner uses a different Kernel Request to determine the next task (if any) to be terminated: Get Next Terminated Task. In this way, controlled access to the Terminate List is maintained by the Kernel, rather than by any other module.

Initiate Task

A task is initiated by linking it to the Ready List. The Task Number of the task to be initiated and the Kernel Request Information Area must be passed along as data with the Interrupt.

The task indicated in the Information Area is linked into the Ready List, based on its Operating Claim. The Information Area is a 40-digit field located at address 8000-8039 of the Kernel Data Area. (Refer to Table 7-4, Kernel Data Area Request Information Area.)

Change Task Priority

The Kernel can be requested to change the priority of a task other than the current task. This is accomplished by changing the Operating Claim of the task. (In contrast to Change Task Priority, Change My Priority is the Kernel Request to be invoked for changing the current task's priority.) The action to be taken in this request depends to a large extent on the current state of this task.

If the task is running, it must be delinked from the Ready List. The Priority and Operating Claim fields of the task's Reinstate List entry must be changed. Then the task must be relinked into the Ready List based on the new Operating Claim.

If the task is dozing, is waiting on an event, or has failed, then it is not in the Ready List, and only the Priority and Operating Claim fields need to be changed.

If the task to be changed is terminating, then the priority change is ignored.

If the task is waiting on a Lock, then its Priority and Operating Claim must be changed. Any task that is delaying the original task has an Operating Claim that in turn reflects the sum of all waiting tasks' Operating Claims. Thus, all of those lock owners must have their Operating Claims adjusted too. If the last task in the lock owner's list is running, it must be relinked into the Ready List based on its new Operating Claim.

Suspend I/O Independent Runner

The I/O Independent Runners are invoked directly by the Kernel. When these Independent Runners have finished processing all of the I/O interrupts, they must inform the Kernel to stop reinstating them.

The Kernel is then requested to suspend the current task, which is one of the I/O Independent Runners, until an I/O Complete has occurred. It is delinked from the Ready List.

Get Next Failed Task

This request is used by the System Failure Independent Runner to determine if there are any more failed tasks to process. The routine simply delinks the next task from the head of the Failure List and returns the Task Number in FHC-TK, a system table entry. If there are no more tasks in the Failure List, then the independent runner is suspended until the next task fails.

Get Next Terminated Task

This request is used by the Task Terminate Independent Runner to determine if there are any more terminated tasks to process. This routine simply delinks the task at the head of the Terminate List and returns the Task Number in the TRM-TK, a system table entry. If there are no more tasks to terminate, the independent runner is suspended until the next task fails.

Time Change

When a time change has occurred, the Kernel must re-evaluate the wake up times of tasks in the Doze List. The request is used by the Keyboard Command processing code for the keyboard commands TR (Time Reset) and DR (Date Reset) in order to change the processor's internal Time-of-Day timer.

The Keyboard Command processing code must pass the Kernel Request Information Area the new date data with the Interrupt instruction.

System Initialize

This request is only used at the end of a bootstrap load of the operating system. All internal list pointers are initialized and a call will be made to the routine to create System Initialize (Coldstart).

The routine will call the Create Independent Runner routine that will in turn call the Memory Manager to create the System Initialize Routine (that is, Coldstart/Halt Load). On return from the Create Independent Runner routine, the Kernel will automatically link the System Initialize Independent Runner task into the Ready List.

Give Independent Runner's Task Number

This request is used by the Create Independent Runner routine to identify the Task Number of any special task that is dispatched directly out of a suspended state. Such special tasks are not invoked directly from the Ready List.

The Create Independent Runner routine passes the Kernel Request Information Area with the Independent Runner Code and the Task Number of the independent runner just created. (See Section 8 for a list of the Independent Runner Codes.)

This routine calculates the relative Reinstate List entry address for the Task Number passed. This address is then stored in the appropriate variable in the Kernel's Data Area.

Midnight Time Change

This request is used only by the Midnight Update Independent Runner, which dozes until a few milliseconds past midnight. At midnight it will invoke the Midnight Time Change routine to do Doze List maintenance and reset the Time-of-Day timer. The Midnight Time Change routine resets the time portion of the timer to zeros and updates the wake up times for those tasks in the Doze List.

Add Channel to SRD Chain

This routine adds the new channel to the end of the appropriate (Normal or Real-Time) Scan Result Descriptor (SRD) list. If the exchange number is non-zero, the channel is also linked onto the end of the indicated exchange.

The calling task passes to the Kernel Request Information Area the new channels exchange, the new channels to add, and an indicator as to whether it is a Normal or Real-Time device.

Delete Channel from SRD Chain

This routine requests the Kernel to delete a channel from either the Normal or Real-Time SRD list. The calling task should pass the Kernel Request Information Area, with the channel to delete and an indicator as to what list the channel is in. A channel can also be removed from the exchange with this request.

This routine does not do any maintenance on the Channel entry table, the IOAT entry, or the EU table entry that can be affected by the deletion of a channel. Any I/Os that are queued against the channel must be handled prior to invoking this routine.

Quiesce/Unquiesce the System

The Kernel can be requested to bring the system to a quiescent state. This request is used at Halt/Load time so that no system tasks will run during the Halt/Load (except the I/O independent runners and IDLE).

Change My Priority

The Kernel can be requested to change the priority of the current task. This routine is invoked at the end of BOJ so that the task can change its priority to the proper value. The current task should pass the Kernel Request Information Area with the new priority as data to the interrupt.

After the task has the Priority and Operating Claim fields of its Reinstate List entry updated, it is relinked into the Ready List based on the new value. This request, unlike the Change Task Priority request, is simplified by the fact that the task being updated is runnable.

Relinquish Processor

This request tells the Kernel that the current task has nothing to do. This routine will relink the current task onto the end of its Operating Claim level. If no other tasks are in this level and there are no user tasks with a higher priority, then the request has no effect.

Alter Interrupt Mask

This request tells the Kernel that the current task wishes to alter its interrupt mask. Once the interrupt mask in the Reinstate List entry has been changed, the task will run with the new mask when it is next dispatched.

Get New Time Slice

This routine is almost identical to the Timer Interrupt routine. The requesting task's direct time will be incremented by the amount of time last used and the task will receive a new time slice. (See Section 9, The Maskable Interrupt Requests.)

Run Normal I/O Independent Runner

The Kernel is requested to run the Normal I/O Complete Independent Runner. This routine is used to support the restarting of an Exchange Queue through the Restart R/D or to catch an overtime I/O R/D. If not already linked, this Independent Runner will be linked into the Ready List.

Run Real-Time I/O Independent Runner

The Kernel is requested to dispatch the Real-Time I/O Complete Independent Runner. This request is used to support the restarting of an Exchange or to catch an overtime I/O R/D. If not already linked, the Independent Runner will be linked into the Ready List.

IDENTIFYING MCP KERNEL REQUESTS

The MCP Kernel Requests are invoked because the operating system executes an INT instruction. The value (06) resides in the Interrupt Cause Descriptor byte and the MCP Kernel Request code is found in the MCP Kernel Request Code field of the Kernel Data Area.

Table 7-3. MCP Kernel Requests and the Associated Kernel Request Codes

MCP KERNEL REQUESTS	KERNEL REQUEST CODE
(Reserved)	00
Doze	01
Awaken Dozing Task	02
Adjust Dozing Period	03
Terminate Current Task	04
Initiate Task	05
Priority Change	06
Suspend Special I/O I.R.	07
Get Next Failed Task	08
Get Next Terminated Task	09
Time Change	10
System Initialize	11
Give I.R.'s Task Number	12
Midnight Time Change	13
Add Channel to SRD Chain	14
Delete Channel to SRD Chain	15
Quiesce the System	16
Change Current Task's Priority	17

Table 7-3. MCP Kernel Requests and the Associated Kernel Request Codes (cont)

MCP KERNEL REQUESTS	KERNEL REQUEST CODE
Relinquish Processor	18
Alter Interrupt Mask	19
MICR Request	20
Run Normal I/O I.R.	21
Run Real-Time I.R.	22

Table 7-4. Kernel Data Area

ABSOLUTE MEMORY ADDRESS	PURPOSE
00 - 39	Indirect Field Length
00 - 07	Undefined
08 - 15	Index Register 1 (IX1)
16 - 23	Index Register 2 (IX2)
24 - 31	Index Register 3 (IX3)
21 - 22	Interrupts Occurred Code
32 - 33	Instruction Interrupt Cause Descriptor
34 - 35	MCP Kernel Request Code
38 - 39	SCAN Result Storage
40 - 45	Kernel Stack Pointer
46 - 47	Breakpoint Pattern for Kernel
48	HALT Execution Digit
49	Internal I/O Mask
50 - 71	Unused
72 - 81	R/D Storage Area
82 - 85	Current Task Number
86 - 93	Reserved
94 - 99	Kernel Interrupt Branch Address
100 - 111	Channel "00" Result Descriptor and Link
120 - 131	Channel "01" Result Descriptor and Link
.	. . .
.	. . .
.	. . .
1640 - 1651	Channel "77" Result Descriptor and Link
8000 - 8039	MCP Kernel Request Data

SECTION 8
INDEPENDENT RUNNERS

GENERAL

Several routines, termed Independent Runners (I.R.s), are used to perform the many system status change, reporting, and housekeeping functions. In order to keep the operating system functioning efficiently and smoothly, some Independent Runners are scheduled to run at various intervals. The individual Independent Runners, their functions, periodicity, required data structures, and how they are managed will be discussed in this section.

There are three routines associated with I.R. manipulation:

- Create I.R.,
- Terminate I.R., and
- Awaken I.R.

Create I.R. code oversees the creation of the I.R., while Terminate I.R. allows the I.R. to terminate itself. Awaken I.R. awakens an I.R. from dozing.

I.R. HANDLING ROUTINES AND THEIR DATA STRUCTURES

All Independent Runner handling routines are driven with the use of two tables:

- Independent Runner Definition Table and
- Active Independent Runner Table.

These tables are stored in the Global Module of the operating system.

I. R. DATA STRUCTURES

Independent Runner Definition Table

This table defines all Independent Runners that can be created on the system, the code module that must be present for the creation of the I.R., and the I.R.'s attributes. It also keeps track of how many copies of the I.R. are currently active and whether or not multiple copies of the I.R. are permitted.

The following is an overview of the entries contained in the Independent Runner Definition Table:

- I.R. Name
- Information necessary for execution, once the I.R. is created:

  Initial instruction address

  Initial active page table

  Initial mode toggles

Initial internal state toggles

Initial interrupt mask

Initial priority

- I.R. Time Slice value
- Environment Number of the Code Segment of the I.R. — where the "User Code" for the I.R. is located
- Number of pages for the I.R. Data Area — the "User Mode" Data Area
- Module name
- Flags, including:

  IRWAKE enable

  Multiple copies of this I.R. allowed indicant
- Number of copies of this I.R. currently active

Active Independent Runner Table

This dynamic table contains an entry for each I.R. active on the system, and the associated Task Numbers. There are as many entries in this table bearing the same I.R. name as the counter value in the I.R. Definition Table indicates are currently active.

The following is an overview of the entries contained in the Active Independent Runner Table:

- I.R. Name
- I.R. Task Number

I.R. HANDLING ROUTINES

Create I.R. Routine

This routine creates a copy of the I.R., provided that the Multiple Copies Allowed field in the I.R. Definition Table is set.

Once called, this module obtains a Reinstate List entry and allocates the Environment Table for the I.R. and initializes it. It will also allocate the User Service Memory Area Table (USMAT) and initialize it. Memory space will be allocated to the Data Area, which will be initialized. Then the initial state for the I.R. will be entered in I.R.'s Reinstate List entry.

The newly created I.R. will be entered into the Active I.R. Table, and the number of active copies of that I.R. will be incremented in the I.R. Definition Table.

A valid request for the creation of an I.R. will always be granted, even if it is necessary to do a Rollout in order to accomplish this.

Terminate I.R. Routine

This module allows an I.R. to terminate itself. It can be called only by an I.R. that wishes to terminate itself. The I.R. will be removed from the Active I.R. Table, and the count of active copies in the I.R. Definition Table will be decremented.

The ultimate function of this routine is to return all memory and data structures allocated for the terminating I.R. back to the system.

Awaken I.R. Routine

This module awakens a dozing I.R. and causes it to run prior to the completion of its doze time. In order for this to actually be possible, the I.R. must have its Early Wakeup bit set (enabled) in the I.R. Definition Table.

This routine does not transfer control back to the caller. Rather, after awakening the I.R., it performs an interrupt in order to give the I.R. the chance to execute.

INDEPENDENT RUNNERS AND THEIR FUNCTIONS

Independent Runners perform reporting, housekeeping, and system status functions for the operating system. They are special service tasks that run for the operating system rather than for a user task. Some I.R.s are run periodically in order to keep the system running efficiently and to ensure that certain functions are performed at specified intervals.

Services provided to the operating system by the Independent Runners include:

- I/O processing,
- driver routines,
- Maintenance Log and Run Log management routines,
- memory and resource management, and
- task handling and system maintenance services.

Many of the Independent Runners that perform these functions are discussed next.

I/O Processing Independent Runners

I/O processing I.R.s handle I/O completions on the system. They include the Normal I/O, I/O with Error, and Supplemental I/O I.R.s.

The Normal I/O I.R. handles I/O completions with no exception conditions.

I/O with Error I.R. handles those I/O completions that have exception conditions, and I/O retries that do not require waiting.

I/O completions with errors that require waiting are handled by Supplemental I/O I.R.s.

Driver Independent Runners

I.R.s in this group include the Reader-Sorter, Uniline DLP, Debug OCS, and OCS Drivers, and Command I.R.

The Reader-Sorter Drivers handle all I/Os to MICR and OCR Check Sorters, and run the pocket select routine in the user's sorter handler programs.

The Uniline DLP I.R. performs constant reads of the Uniline DLP in order to transfer data to the operating system.

The Debug OCS I.R. handles communication with the user at the OCS or ODT. It formats the Debug screens, passes Debug commands to the Debug Module, and writes Trace information to the printer or the printer backup.

The OCS Driver I.R. performs constant default reads of the OCS in order to transfer data to the operating system.

Command I.R. is the driver that handles control commands. Its function and operation are covered in detail in Section 13, Command I.R.

Maintenance Log and Run Log Management Independent Runners

I.R.s from this group include The Maintenance Log Transfer and Write I.R.s and the Run Log Transfer I.R.

The Maintenace Log Transfer I.R. closes the existing Maintenance Log (MLOG) file, opens a new MLOG file, and generates system configuration records in the new MLOG file.

Maintenance Log Write I.R. writes MLOG records from the MLOG buffer into the appropriate MLOG file when the buffer becomes full.

Run Log Transfer I.R. closes the existing Run Log (RLOG) file with a pseudo-EOJ record. It opens a new RLOG file with a pseudo-BOJ record.

Memory Management Independent Runners

I.R.s in this group include the Rollin, Rollout, and Resource Manager I.R.s.

Rollin I.R. handles the transfer of information from disk to memory; Rollout I.R. handles the transfer of information from memory to disk. Both of these functions are discussed in detail in Section 4, Memory Area Rollin/Rollout.

The Resource Manager I.R. handles tasks awaiting either memory expansion or the rollin of information from disk to memory. The Message Module notifies the Resource Manager of a waiting task. A table containing each waiting task's memory priority is maintained by the Resource Manager. It calls the Memory Manager in attempt to secure memory for a task, based on its recorded memory priority. For an example of the Resource Manager's role in Beginning of Task processing, see Section 13, Allocation of the Task's Code Area.

Task Handling and System Maintenance Independent Runners

I.R.s in this group include the Terminate Task, Stack Overflow, Idle, Timer, and Status I.R.s.

Terminate Task I.R. is called when a task has gone to EOJ. It calls the Memory Manager in order to return the task's memory for other use. (See Section 7, Terminating List and Terminate Current Task.)

The Stack Overflow fault and Stack Overflow I.R. are discussed in Section 10, Stack Overflow.

The Idle I.R. is dispatched by the Kernel when no other tasks are ready be to dispatched in order to absorb idle CPU time.

Timer I.R. handles the periodic waking of tasks that are waiting on some condition.

Status I.R. handles most housekeeping functions that occur on a periodic basis, including device status and midnight and date changes.

SECTION 9
INTERRUPT HANDLING

INTERRUPT PROCESSING

There are two mechanisms for interrupting an instruction stream in order to start execution of the appropriate MCP routines:

- Interrupt Procedures and
- Hardware Call Procedures.

The Interrupt Procedure is used by the processor hardware to transfer the system environment to the MCP Kernel. A Hardware Call Procedure is executed in response to certain processor-detected faults. While the Hardware Call Procedure will be taken up elsewhere (Section 10, Hardware Call Procedure), we will briefly say a few words about it here.

Once executing, a Hardware Call Procedure changes the system environment and transfers control to a software error handling routine. In a manner similar to the Interrupt Procedure, the system state must be saved for return from the error handling routine. Thus, the state of the processor must be saved in the Hardware Call Stack Frame that is associated with the called routine.

The instruction address of the next instruction to be executed (unless one of the faults that require that the address of the failing instruction be stored is also present) is included in the state information as a result of the following faults: Trace, Programmatic Soft Fault, and other Memory Area Faults.

INTERRUPTS

In the event of an interrupt, the CPU suspends execution of the running task and control is transferred to the MCP Kernel routine, unless the Interrupt Mask is set to ignore the interrupt. The Kernel then services the interrupt.

Whenever an interrupt occurs, the state of the interrupted process is saved, including the program counter, registers, and condition codes. The Kernel then executes the code necessary to service the interrupting condition. Finally, the state of the interrupted task is restored to exactly the same state it was in before the interrupt was invoked. The Interrupt Handler thereby maintains transparency for the executing task.

The task running at the time of the interrupt is not aware of the fact that it has been stopped, saved, and restarted. Other interrupts are transparent to the running interrupt process, therefore any number of interrupts can exist at a given time and can interrupt each other.

MASKABLE AND NON-MASKABLE INTERRUPTS

Interrupts are divided into several categories. Some interrupts are maskable; others are not.

Maskable interrupts can be ignored by the running task, whereas non-maskable interrupts are serviced immediately. The setting of the Interrupt Mask determines whether or not a maskable interrupt will be ignored. The interrupts, both maskable and non-maskable, are listed in Table 9-1.

INTERRUPT MANAGEMENT

In order to facilitate interrupt servicing, the Interrupt Handler relies on three data structures:

- The Interrupt Mask
- The Interrupt Cause Descriptor
- The Interrupt Frame

These three structures aid in resolving interrupts, determining where an interrupt falls in the established interrupt priority scheme, and recording information on the nature and cause of the interrupt.

Table 9-1. Maskable and Non-Maskable Interrupts

INTERRUPTS	MASKABLE
System Overtemperature	Yes
Timer	Yes
Real-Time I/O Complete	Yes
Normal I/O Complete	Yes
Error I/O Complete	Yes
Interrupt Instruction (INT,OP=90)	No
Failed Lock	No
Failed Event	No
Released Lock	No
Released Event	No
Failed Hardware Call	No

Interrupt Mask

The 8-bit Interrupt Mask determines which of the maskable conditions will be allowed to physically interrupt the current processing. If an interrupt occurs, and the corresponding bit(s) in the Interrupt Mask are set, a physical processor interrupt is enabled.

If an Interrupt Mask bit is reset, any event belonging to that bit's indicant category will not cause an interrupt at that time. Rather, that event will be held pending until an Interrupt Mask is loaded that defines that category as non-maskable.

The Interrupt Mask definitions are listed in Table 9-2.

Table 9-2. Interrupt Mask Definitions

CONDITION	BIT	CAUSE
Reserved	7	
Reserved	6	
Overtemp	5	System Overtemperature
Task Timer	4	MSD = 0
Reserved	3	
Real-Time I/O	2	I/O Complete on a Real-Time Device
I/O Error	1	I/O Complete, Exceptions on a Non-Real-Time Device
Normal I/O	0	I/O Complete, No Exceptions on a Non-Real-Time Device

Interrupt Descriptor

An Interrupt Descriptor is written at locations 21-22 of the Kernel Data Area. Refer to Table 9-3 for the Interrupt Descriptor bit values.

Table 9-3. Interrupt Descriptor Bit Values

CONDITION	BIT	CAUSE
Reserved	7	
Instruction	6	Instruction-related Interrupt
Overtemp	5	System Overtemperature
Task Timer	4	MSD = 0
Reserved	3	
Real-Time I/O	2	I/O Complete on a Real-Time Device
I/O Error	1	I/O Complete, Exceptions on a Non-Real-Time Device
Normal I/O	0	I/O Complete, No Exceptions on a Non-Real-Time Device

Note that the Interrupt instruction (INT, OP=90) is not maskable. However, the other Interrupt Descriptor bits are the logical "AND" of the pending interrupt conditions and the Interrupt Mask. Instruction interrupts are flagged in the Interrupt Descriptor with the actual Instruction Interrupt Condition stored in the Instruction Interrupt Cause Descriptor.

Interrupt Cause Descriptor

The Interrupt Cause Descriptor is stored at location 32-33 of the Kernel Data Area when an unmasked Instruction interrupt occurs. This 8-bit entity stores the actual request being made. Refer to Table 9-4 for the Interrupt Cause Descriptor bit values.

Table 9-4. Interrupt Cause Descriptor Bit Values

VALUE	CAUSE
00	Reserved
01	Failed Lock
02	Failed Event
03	Released Lock
04	Released Event
05	Failed Hardware Call
06	Executed Interrupt Instruction (OP=90)
07	Failed Virtual Branch Reinstate
08 - FF	Reserved

Interrupt Frame

When an interrupt occurs, the interrupt conditions are tested and the interrupt procedure is initiated at the end of the current instruction. The address of the current instruction or the next instruction to be executed upon completion of the interrupt servicing is stored in the Interrupt Frame. Whether the current or the next instruction is stored depends on the type of interrupt that has occurred.

The interrupt state is saved in the Reinstate List entry of the interrupted task. The interrupt state consists of the following information:

- Measurement Register
- Interrupt Mask
- Mobile Index Registers
- COM and OVF Flags
- Mode Indicator
- Soft Fault Enable
- Privileged Indicator
- Trace Indicator
- Snap Enable Indicator
- Active Environment Number
- Instruction address

See Section 4 for a discussion of the Reinstate List.

INTERRUPT PROCEDURE

The Interrupt Procedure is used to enter the MCP Kernel function specified by the 6-digit address located at the Kernel Data Area address 94. The processor registers and state are stored in the Interrupt Frame and control is transferred to the Kernel. The rest of this section describes these steps in greater detail.

1. The current value of the Task Timer is stored into the Time Slice Remaining field of the Reinstate List. This field will be evaluated by the Kernel's Dispatcher. The value of the Task Timer is then reset to the maximum value.

2. The 2-digit Interrupt Descriptor is stored into absolute memory locations 21-22 of the Kernel Data Area.

3. The machine state of the interrupted task is stored in the Interrupt Frame, which is located in the Reinstate List entry for the interrupted task.

4. The interrupt conditions are selectively reset according to the Interrupt Mask. That is, if the bit in the mask is set to "1", then the corresponding condition is reset. If the bit is set to "0", then the corresponding condition will not be reset. The Instruction Interrupt condition is then reset.

5. The machine state is then set according to Table 9-5.

6. The MOPOK signal is set to "0" while the Measurement Register is being changed and "1" at all other times.

7. The Interrupt Mask Register is set to zero.

8. The 4-digit Task Number for Task #1 is stored at the Kernel Data Area address 82.

9. The MAT pointed to by the first entry in the ET for Task #1 is located. This is the Kernel's MAT. Its first eight entries are then loaded into the processor's registers.

10. An Unconditional Branch to the 6-digit address relative to Base #1, located at memory address 94 relative to Base #0 is executed. At this point, if any Hardware Call conditions exist, a REDLIGHT Halt occurs after the fault indicators are stored in absolute memory locations 72-81 (the R/D Storage Area of the Kernel Data Area).

Table 9-5. Machine State Information and Settings

INFORMATION	SETTING
Kernel Mode	SET
Active Environment Number	000000
Current Task Number	0001
Privileged / User Mode	PRIVILEGED
Trace Mode	NON-TRACING
SNAP Enable	DISABLED
Soft Fault Enable	DISABLED
Measurement Register	0000 0000
Comparison and Overflow Flags	RESET

THE MASKABLE INTERRUPT REQUESTS

System Overtemperature Interrupt

This interrupt indicates that the processor is entering a power-down cycle due to overheating. Although this is theoretically a maskable interrupt, the system will not be oblivious to this condition because the Overtemperature Warning bit will have been set in the periodic System Status Instructions' Result Descriptor.

This interrupt puts the Kernel into an infinite loop. The Dispatcher never has a chance to run and no more I/Os can be initiated or serviced.

Timer Interrupt

This maskable interrupt indicates to the Kernel that the Time Slice for the current task has been exceeded. The Kernel is requested to give a new Time Slice to the task, perform the time accounting functions for that task, and determine if there is another task of equal or greater Operating Claim that should be dispatched. (See Section 11, Dispatcher.)

The Timer Interrupt is invoked by the hardware whenever the most significant digit of the Time Slice Remaining Register is "0" and the Time Slice Interrupt Enabled bit of the Interrupt Mask is enabled. (See Section 9, Interrupt Mask.)

Upon invocation, the Time Slice Remaining field of the current task's Reinstate List entry is incremented by the Time Slice Increment field. The Direct Time Used field is also incremented by this amount. The current task's Reinstate List entry is then relinked into the Ready List based on the task's Operating Claim. (See Section 11, Dispatcher, Section 4, Ready List, and Section 4, Reinstate List.)

Whenever a Timer interrupt occurs, the RE-TIM bit in the task's Reinstate List entry must be examined. If this bit is set, then the task must treat the Timer interrupt as a fault rather than as an interrupt, via the Soft Fault mechanism. (See Section 10, Fault Handler and Section 10, Soft Fault.)

Real-Time I/O Interrupt

This request indicates that an I/O has been completed on a real-time device. The Kernel is requested to link the appropriate Real-Time I/O Complete Independent Runner into the head of the Ready List. (See Section 8, Independent Runners.)

Normal I/O Complete

This interrupt indicates the completion of an I/O with no errors on a non-real-time device. It causes the Normal I/O Complete Independent Runner to be executed.

The Reinstate List entry associated with the Normal I/O Complete Independent Runner is linked to the head of the Reinstate List. The Task Number of the Normal I/O Independent Runner must be passed to the Kernel through a Kernel Request.

Error I/O Complete

This interrupt indicates the completion of an I/O with errors on a non-real-time device. It causes the Error I/O Complete Independent Runner to be executed.

The Reinstate List entry associated with the Error I/O Complete Independent Runner is linked to the head of the Reinstate List. The Task Number of the Error I/O Independent Runner must be passed to the Kernel through a Kernel Request.

THE NON-MASKABLE INTERRUPT REQUESTS

These interrupts indicate that a Firmware or Software Kernel Request has been made. The Interrupt Cause Descriptor must be examined in order to determine the actual request that was made. Since they are non-maskable, these requests occur regardless of the Interrupt Mask settings. (Refer to Table 9-4, Interrupt Cause Descriptor Bit Values.)

Interrupt Instruction

An MCP Kernel request is made with the Interrupt Instruction (OP=90). This instruction writes the specific Kernel request in the 2-digit Kernel Data Area location 34. The individual Kernel requests are discussed in Section 7 and the Interrupt Instruction is discussed in Appendix B.

Failed Lock

This non-maskable request indicates that the hardware has attempted to perform an Unconditional Lock variant of the Lock instruction against a lock that another task already owns. The firmware requests the Kernel to suspend the current task until the lock is released and then links this task into the Lock List.

The Kernel removes it from the Ready List and then increments the Operating Claim of the task that owns the lock by the Operating Claim of the current task. If that task itself is waiting on a Lock, then the incrementation of Operating Claim is propagated onto the task that it is waiting on. This process continues until the Operating Claim of a task not waiting on a Lock is incremented. (See Appendix B, LOK instruction.)

Failed Event

This non-maskable request indicates that the firmware has attempted to perform a Wait variant of the Lock instruction against an event that has not been caused. The Kernel is requested by the firmware to suspend the current task until the Event on which this task is waiting is caused by another task.

The firmware links the task into the appropriate Event List and the Kernel routine simply removes the current task from the Ready List. In fact, the task is removed from the Ready List regardless of any other Kernel requests that may occur simultaneously. The other requests will be processed, but they will not put the task back into the Reinstate List. (See Appendix B, LOK instruction.)

Released Lock

This request signals that the firmware just completed the execution of an Unlock variant of the Lock instruction against a Mutual Exclusion Lock that has other tasks suspended. The Kernel is requested to link the waiting tasks into the Ready List and to adjust the Operating Claim of the current task to reflect the fact that this task is no longer blocking the other tasks.

This Kernel routine is not invoked unless there are tasks waiting for the lock to be released. If there is no contention for the lock, then the Unlock variant simply releases the lock and no interrupt occurs.

The firmware provides the Kernel with the address of the Reinstate List entry at the head of the Lock List for that lock. This address is stored in the processor's IX3 register.

The Lock List is traversed and the following two functions are performed for each entry:

- The Operating Claim of the current task is decremented by the the Operating Claim of the Reinstate List entry that is being processed. This indicates that the current task is no longer blocking out the task that is being awakened.
- The Reinstate List entry for the task that is being awakened is linked into the Ready List, based on the value of its Operating Claim.

(See Appendix B, LOK instruction.)

Released Event

This request signals that the firmware completed the execution of the Cause variant of the Lock instruction against an event that has tasks suspended. These tasks are waiting for the event to be released. The Kernel is requested to link those suspended tasks into the Ready List.

Like the Released Lock, the Released Event does not occur if there were no tasks waiting for the event to be signalled. If there was no contention for the Event, then the Cause instruction just releases the Event Lock and no interrupt occurs.

The firmware stores the address of the Reinstate List entry at the head of the Event List for that event in the processor's IX3 register. The Event List is then traversed, and each Reinstate List entry in the Event List is linked into the Ready List, based on the task's Operating Claim. (See Appendix B, LOK instruction.)

Failed Hardware Call

This request indicates that the firmware just attempted to execute a Hardware Call Procedure for the current task. Since this usually indicates that a fairly severe system error has occurred, the Kernel is requested to suspend the current task and to invoke a special System Failure Independent Runner to examine the failure.

The current task is removed from the Ready List and is marked as waiting for attention from the System Failure Independent Runner, which is then linked into the Ready List.

SECTION 10
FAULT HANDLING

FAULT CLASSIFICATIONS

Faults are classified into two categories:

- Soft Faults and
- Hard Faults.

They are distinguished by the state that the Hardware Call Procedure saves.

Soft Faults are the faults that will allow execution of the next instruction upon recovery from the current error. When a Soft Fault occurs, the next instruction's address is saved in the state information.

Hard Faults require the current instruction to be re-executed after successful error recovery. Thus, in contrast to the case of a Soft Fault, the address of the current instruction is saved.

The following list shows which category the various faults belong to. The faults are covered individually at the end of this section.

SOFT FAULTS

Soft Memory Area Fault
Software Fault
Trace Fault

HARD FAULTS

Invalid Arithmetic Data
Hard Memory Area Fault
Instruction Timeout
Address Error
Uncorrectable Memory Parity Error
Invalid Instruction
Accumulator Trap
Snap Picture
Stack Overflow

HOW THE FAULT HANDLER IS INVOKED

As we have already noted, a Hardware Call Procedure is executed to enter the Fault Handler Routine when the processor detects a fault. A software program cannot directly call the Hardware Call Procedure.

In a manner similar to the Interrupt Procedure, the system state must be saved for return from the error handling routine. Thus, the state of the processor must be saved in the Hardware Call Stack Frame associated with the called routine.

The instruction address of the failing instruction is included in the state information when the following faults have occurred: Address Error, Invalid Instruction, Invalid Arithmetic Data, Invalid Alter Table Entry, Accumulator Trap, Uncorrectable Memory Errors, Instruction Timeout, and certain Memory Area Faults.

The instruction address of the next instruction to be executed (unless one of the faults that require that the address of the failing instruction be stored is also present) is included in the state information as a result of the following faults: Trace, Programmatic Soft Fault, and other Memory Area Faults.

THE HARDWARE CALL PROCEDURE

The operations that the Hardware Call Procedure performs are discussed as follows.

1. The 10-digit Fault Indicators (the Processor Result Descriptors) are stored at memory address 72-81 relative to the MCP Data Area of the currently executing task.

2. If the Interrupt Mode toggle is set, indicating that we are in Kernel Mode, the processor will halt. Otherwise, the following steps occur.

3. The Hyper Call Function Descriptor Table is located by using the Base and Limit addresses stored at locations 87-92 and 94-99 respectively of the current task's MCP Data Area. The 20-digit Function Entry containing the addressing environment of the Fault Handler Module is then obtained from the Function Descriptor 0 of this table. Note that the Hardware Call is, in essence, a forced Hyper Call (HCL) to function 0 of the current task except that Hyper Call 0 does not have the Fault Indicators.

   The Hyper Call Function Entry contains the following information:

   - Environment Number,
   - Next Instruction Address,
   - Protection Field,
   - Interrupt Mask, and
   - Mode Indicators.

   The new addressing environment for HCL 0 is the one in which the Fault Handler Module will run. The next instruction address in the Function Descriptor is the entry point address to the Fault Handler Routine.

4. The processor's state, in this new addressing environment, is saved on the Hardware Call Stack Frame. Entries for both Soft and Hard Faults are essentially the same, except for the next instruction address, as noted earlier.

5. The state of the processor is changed to the one obtained in step 3, and execution begins at the address indicated in the Function Descriptor 0.

If any faults occur during the execution of the Hardware Call Procedure, it is termed a Failed Hardware Call. In this instance, the new Fault Indicators are stored in the task's Reinstate List entry in the Failed Hardware Call Result Descriptor Area. A Failed Hardware Call code value is stored in the State Indicator field. An Interrupt Cause Descriptor is stored at location 32-33 of the Kernel Data Area and the processor will then generate an interrupt. The Interrupt Procedure takes over from there and the task will be terminated eventually.

FAULT HANDLER MODULE

This module contains code that analyzes and handles each fault condition. The Fault Handler Routine and the Memory Manager's Memory Area Fault Routine must always reside in memory.

The Fault Handler Routine is executed by the Hardware Call Procedure with the following attributes stored in the task's Function Entry 0.

- Environment Number - Memory Area Table containing the Fault Handler Routine
- Next Instruction Address - of the Fault Handler Routine
- New Interrupt Mask
- New Mode Toggles :
  Privileged
  Soft Fault Disabled
  Trace Disabled
  Snap Disabled

The MAT that the Fault Handler is running with must contain the following Memory Areas:

MEMORY AREA NUMBER	MEMORY AREA
0	Task's MCP Data Area
1	Fault Handler Module
2	Unused
3	Unused
4	Unused
5	Unused
6	Unused
7	System Tables Memory Area

The remainder of the Fault Handler routine tests the faults and performs recovery operations if possible. Since more than one fault can be detected at one time, the following strategy is applied in assigning priority to the faults.

1. If none of the Fault Indicators are set, then it is presumed that a Hyper Call 0 was performed. The task is then terminated.

2. If there is a Hard Fault, no attempt to process Soft Faults is made except the Trace Fault. This is because the faulted instruction will be re-executed, and the same Soft Faults will occur and be handled at this time. If a Trace Fault occurs with a Hard Fault, any action taken for the Hard Fault will be recorded.

3. The detection of Hard Faults is resolved on a priority basis. Once the highest priority Hard Fault is determined, the recovery attempt is made. If recovery is not successful, the task is terminated, and the remaining faults (if any) may be reported if the task has been traced.

4. If a Hard Fault is recovered from successfully, then proceed to step 6 in order to return to the user program.

5. If only Soft Faults are detected, then all faults must be handled because there is no built-in second chance for servicing them as there is with the Hard Faults.

6. After all of the faults have been serviced, a Return instruction (RET, OP=63) is executed in order to return back to the task.

The order in which faults are serviced is as follows:

Stack Overflow
Snap Picture Taken
Memory Parity Error
Invalid Arithmetic Error
Instruction Timeout
Address Error
Invalid Instruction
Hard Memory Area Fault
Accumulator Trap
Soft Memory Area Fault
Software Fault
Trace Fault

HARD FAULTS

Invalid Arithmetic Data

This fault indicates that an undigit other than a sign digit has been detected in an arithmetic operand. There is no recovery for this fault. The task is terminated unless it is armed.

Hard Memory Area Fault

This fault indicates that the instruction is unable to proceed because of an absent memory area, and the instruction needs to be re-executed after the memory area is made present. There are three situations in which this fault can occur.

- Memory Area 0 of the new environment is missing in the case of the HCL or BCT instructions. Since the state and parameters must be stored in the new page table's stack, the memory area must be brought in before re-execution can begin.

- The memory area containing the string for MVS, CPS, and HSH instructions is missing.

- The memory area containing the buffer on which the I/O is to be performed is missing.

The first situation should not occur since Memory Area 0 is non-rollable. However, if it does occur, the task can only be terminated. (The Memory Manager cannot be called, due to the fact that there would be no stack for address 40 of Memory Area 0 to point to.)

In the other two cases, the recovery for a Hard Memory Area fault is the same, whether it occurs in the MCP code or user code. The recovery steps are discussed as follows.

First, the Memory Area Status Table (MAST) entry number of the absent entry are passed as parameters on the Hardware Call Stack Frame by the hardware. If there is more than one missing memory area in the operands, the processor will report only the first one that it encounters. The remainder of the missing memory areas will be repeatedly handled after the task is returned to and the faulted instruction is re-executed again.

The Memory Area Fault routine picks up the Hyper Call parameters and calls the Memory Manager's Rollin routine to attempt to make the missing memory area present. The three possible outcomes from the Rollin routine are discussed below.

- The outcome may be successful, in which case the memory area is made present and the task is returned to.
- If there is insufficient memory, an interrupt will be generated to put the task into a doze state. The next time that the task is dispatched, the Fault Handler will call the Memory Manager and try to bring in the memory area again until it is rolled in.
- If the memory area has parity errors then the task is terminated.

  Refer to the following topics in Section 4 for related information on the Fault Handler: Faulted Entry, Memory Area Address Resolution, and Memory Area Rollin/Rollout.

Instruction Timeout

This fault indicates that the instruction did not complete within the time period specified by the processor. An endless loop of indirection or a search of a very long list are examples of causes. No recovery can be made for this type of error. The Instruction Timeout fault will not occur on a legitimate instruction.

Address Error

This fault indicates that one of the following possible conditions exists:

- the address is less than the base or is greater than the limit,
- a non-decimal digit is contained in the instruction address other than the address controller or the extension digit,
- an odd instruction address has been resolved,
- an Environment Number is invalid,
- there is an invalid descriptor address relationship, or
- there is an Environment Number and Memory Area Number limit error.

Uncorrectable Memory Parity Error

This fault indicates that a parity error has been detected. The processor does not supply the address where the error occurred.

The Fault Handler first interprets the faulted instruction in order to obtain all of the memory areas referenced by the instruction. Then a Memory Manager routine is called to mark all of these memory areas as absent and corrupted. Any subsequent reference to one of these memory areas will result in a Memory Area fault, and the referencing task will be terminated by the Memory Area Fault routines.

After the tagged memory areas are removed, a routine is invoked to put a bad memory record in the configuration file. The next time the system is halt loaded, those areas will be excluded from the available memory list.

Invalid Instruction

This fault indicates that one of the following situations has arisen.

1. A privileged instruction is being executed in nonprivileged mode.
2. The protection field of a Hyper Call or BCT instruction does not have the value "DD".
3. Invalid instruction usage is being attempted.
4. A literal is not allowed.
5. There is a non-decimal value for an indirect field length.
6. Copy is not enabled in the ATE instruction.
7. An invalid opcode has been detected.

When the Invalid Instruction Fault indicator is set, the Invalid Instruction Extension Byte (IEX) will further document the error. A value of "01" in the extension byte indicates that the fault belongs in one of the categories above from (1) to (5).

Accumulator Trap

An Accumulator Trap is caused by either:

- an overflow during the execution of a fixed point instruction, or
- an overflow, underflow, or division by zero in a floating point instruction.

When one of the above situations exists and the Trap Enable field (address 64-65 of the Task's User Data Area) contains the value "FF", the Hardware Call Procedure is executed to enter the Fault Handler. The Fault Handler obtains the Trap Address from address 66-71 of the Task's Data Area. It then stores the 6-digit next instruction address at the location specified by the Trap Address. The task is returned to at Trap Address+12.

Snap Picture

When this bit is set in the Fault Indicators, it indicates that a Snap Picture has been taken. This usually occurs when the Snap Enable Toggle was set and one of the following faults occurs:

- Invalid Arithmetic Data,
- Instruction Timeout,
- Address Error, or
- Invalid Instruction.

The service routine for this fault will log the Snap Picture in the Maintenance Log.

Stack Overflow

A 500-digit filler buffer is kept at the top of the stack. This fault indicates that an attempted stack operation will cause the Top of Stack (TOS) pointer to point within this buffer's range. This fault can be incurred only by one of the following instructions:

- Virtual Enter (VEN),
- Hardware Call (HCL), or
- Adjust Stack Pointer (ASP).

If the operation would cause the TOS pointer to point within the 500-digit buffer, then it is faulted. The address of this instruction is saved so that it will be re-executed upon return from the Fault Handler.

The Fault Handler has an identity passing interface with the Stack Overflow Independent Runner. Therefore, the Task Number of the faulted task is available for the Stack Overflow I.R.

Stack Overflow I.R. processes one stack overflow at a time. It calls the Memory Manager to expand the MCP Data Area of the task whose Task Number it was passed.

SOFT FAULTS

Soft Memory Area Fault

This fault indicates that a memory area was missing when a MAT was loaded. Only the following instructions can cause this fault:

- VEN - Virtual Enter,
- HCL - Hyper Call,
- BCT - Branch Communicate,
- BRV - Virtual Branch Communicate, and
- RET - Return.

The algorithm employed to handle this fault is quite similar to that of the Hard Memory Area Fault. Multiple missing memory areas are handled one at a time, similar to the way multiple missing memory areas are handled in the Hard Memory Area Fault case.

If there are multiple missing memory areas, the processor will report only one at a time. After the first missing memory area is brought in, a RET instruction is executed in order to transfer control back to the task. If there are other missing memory areas, a Soft Memory Area Fault will again occur, this time on the RET instruction. This procedure is repeated until all required memory areas are brought in.

Software Fault

A software Fault is detected when the Soft Fault Enable Toggle (found in the Reinstate List entry for the current task) is set and the Soft Fault Pending Flag is not equal to zero. This fault can be caused by the Return (RET) and Virtual Branch Reinstate (BRV) instructions.

The Software Fault enables a task to queue work to be executed at a later time for itself or for another task. The Fault Handler reads and services all of the task's asynchronous messages.

Trace Fault

This fault indicates that the processor was running in trace mode and that the instruction previously executed should be traced. The Fault Handler will invoke the appropriate Trace routine to handle the fault.

SECTION 11
DISPATCHER

GENERAL

The Dispatcher, like the Interrupt Handler, is a part of the Kernel. The function of the Dispatcher is to distribute processor time equitably to all tasks by use of a priority level assignment scheme.

The Dispatching Algorithm performs two discrete functions:

- It links tasks into the Ready List by use of the Dispatching Key.
- It identifies the next task to be dispatched by reinstating the task at the head of the Ready List.

OPERATING CLAIM

A task's priority changes throughout its life on the system. An initial Operating Claim is assigned to the task at BOJ. (See Section 13, Task Creation.)

Because a task can accumulate many locks during execution, it then is viewed as a processing impediment or bottleneck to other tasks. As mentioned, the Operating Claim of such a task will reflect this condition.

A lock owner's Operating Claim is incremented when some other task fails the lock owner's lock. The owner's Operating Claim is then incremented until the owner unlocks the lock.

DISPATCHING KEY

The Dispatching Key is what determines where in the Ready List a task will be linked. It consists of two parts:

- the Operating Claim and
- the Next Scheduled Run Time.

A task's Operating Claim is, in effect, a measurement of the demand by other tasks that this task be processed as soon as possible. A task that possesses a lock on which many tasks are waiting will have a high Operating Claim.

The Next Scheduled Run Time is the time at which the task is next scheduled to be run. This field is located in the Reinstate List entry for the task.

HOW THE DISPATCHER WORKS

Each job initially will be assigned a static Time Slice, depending on its Operating Claim. When a job has exhausted this Time Slice (Time Slice Exceeded), the processor's Timer Interrupt is issued. (See Section 9, Timer Interrupt.)

At this point, the Kernel will update the Direct Time Used field and will give the task a new Time Slice. The task will then be linked on to the end of its Operating Claim level. Other tasks within this Operating Claim level will then be given the oportunity to be dispatched. Thus, the Dispatching Algorithm utilizes a Round Robin scheme within the Operating Claim levels. Refer to Figure 11-1 for a depiction of the dispatching algorithm.

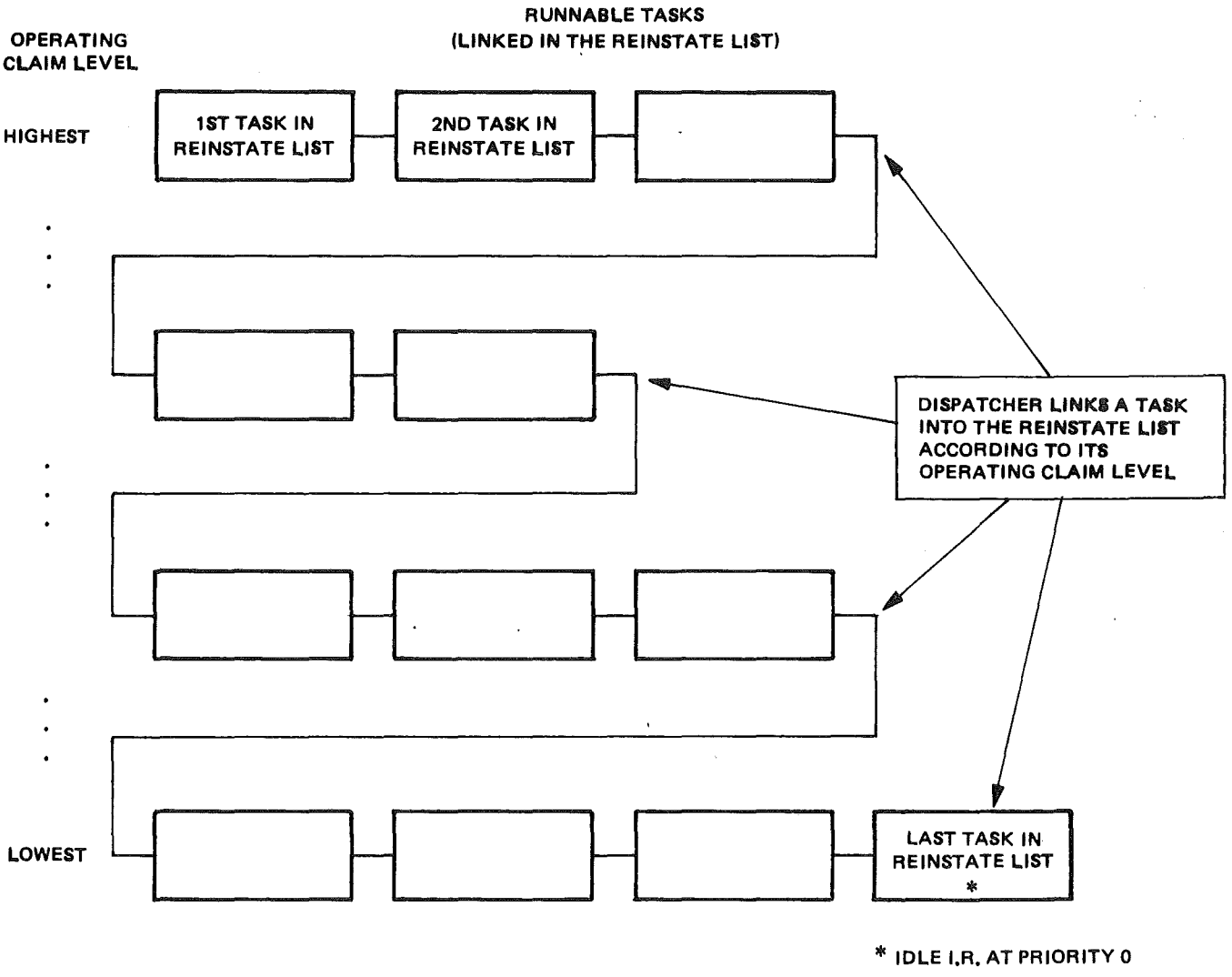

Figure 11-1. The Dispatching Algorithm

SECTION 12
VIRTUAL I/O

GENERAL

Only the most primitive level of the operating system need be concerned with the low level details necessary in performing input/ouput operations (I/Os) on the various physical devices. Concentration of device-dependent information into the primitive layer of the MCP (an intelligent I/O Subsystem) allows the rest of the operating system to communicate with the primitive level using virtual I/O. Any changes to or additions of devices to the system requires that only the primitive layer be updated.

For example, a high level function in the MCP might do a generic Write to some device. The Write will then pass a generic I/O Descriptor to the MCP Virtual I/O Module. The Virtual I/O Module then converts the generic I/O Descriptor into a hard I/O Descriptor that the physical device understands.

QUEUE ELEMENTS

It is not necessary to rebuild the I/O Descriptors and the associated information for the Initiate I/O instruction every time an I/O is initiated. The I/O Queue Element makes this possible.

The I/O Queue Element is a structure that is built for each buffer of a file at the time a file is opened. It contains most of the relevant information about the I/O. Only variable parts of the Queue Element need to be changed before an I/O can be fired.

Thus, the I/O Path under Virtual I/O needs to have multiple entry points in order to handle the various situations such as:

- building the Queue Element,
- changing the Virtual I/O Descriptor, and
- firing the previously converted Virtual I/O Descriptor.

VIRTUAL I/O DESCRIPTOR

The Virtual I/O Descriptor consists of two parts:

- the Virtual Command Descriptor and
- the Relative Buffer Descriptor.

The Virtual Command Descriptor describes the I/O Operation that the user wishes to perform.

The Relative Buffer Descriptor describes the buffer memory that is to be used in the I/O Operation. Because not all of the I/O Operations require a buffer, the Relative Buffer Descriptor is not always necessary.

VIRTUAL COMMAND DESCRIPTOR

The Virtual Command Descriptor is required for all I/O Operations. It defines, in a device-independent manner, the I/O Operation that the user is requesting.

The Virtual Command Descriptor contains three fields:

- an operation code field,
- a variant flag digit field, and
- an extension field.

Each of the fields is described below.

Operation Code Field

The types of operations that are supported are listed below, along with some examples. Other fields in the Virtual Command Descriptor give more information about the operation, and they are discussed later.

- Test or change the status of a device

  (Determine the DLP and status, power the device up or down, cancel the device)

- Position the designated device and transfer no data

  (Skip forward or in reverse, rewind, advance to the end of the media.)

- Write to a device - data is written to the specified buffer address

  (Write data)

- Perform specialized data transfer to a device or its controller - the buffer address is required for most operations

  (Write to the DLP Buffer, load firmware to a controller, initialize the medium with the indicated buffer, relocate sector to the designated spare sector, erase all data to the End of Media)

- Perform write and position operations

  (Write and then position, position and then write)

- Read from a device

- Read status information from a controller - the buffer address is required

  (Read the DLP buffer into memory, Read the controller buffer, read the extended R/D from the DLP into the specified buffer, place the unit status into the buffer, read extended status information from device.)

- Write and then read

  (Write to a device and then perform a read of the device, write to DLP Buffer and then read from the buffer)

Variant Flag Digit

The variant flag digit is used in combination with the operation to indicate:

- whether or not the operation can be timed out,
- whether the operation will be performed normally or in reverse,
- if data is to be transfered to the device, and
- whether or not the data is translated from EBCDIC to ASCII, or from ASCII to EBCDIC.

Extension Field

The extension to the Virtual Command Descriptor is used to indicate the device (by its Unit Number) and the sector address of the device at which the I/O operation begins.

RELATIVE BUFFER DESCRIPTOR

The Relative Buffer Descriptor describes the buffer in memory that is to be used in the I/O operation, provided that it is required by the operation. The Relative Buffer Descriptor contains three fields:

- the Environment Number,
- the Memory Area Number, and
- buffer begin and end addresses.

The Environment Number, Memory Area Number, and Buffer Begin and End Addresses

The buffer is assumed to lie between buffer begin and buffer end addresses in the memory area identified by the Memory Area Number (MAN). The MAN is used as an index into the Memory Area Table in order to locate the memory area that contains the buffer. The Environment Number (EN) specifies the correct MAT.

VIRTUAL RESULT DESCRIPTORS (R/Ds)

Like the Virtual I/O Descriptor, the Virtual Result Descriptor (R/D) has a uniform format. The Physical I/O Complete routine forms the Virtual Result Descriptor from the hardware device's own Result Descriptor.

Both the original DLP R/D and the Virtual Result Descriptor are stored in the Virtual I/O Queue Element. Therefore, the Virtual Result Descriptor needs only to describe the general class of an error, and further details can be found in the DLP R/D.

The Virtual Result Descriptor contains three parts that are described below.

- Error flags

  These flags indicate the common error conditions found in the DLP R/D.

- Additional information

  Further error information about some errors is included (such as where the error occurred).

- R/D data

  Any data present in the R/D is placed here (such as printer type).

VIRTUAL QUEUE ELEMENT

The 400-digit Queue Element contains all information relevant to the performance of I/O. The I/O to be performed, which device, which buffer, and all of the information on the result of the I/O are included in the entry. For the definitive table entries, consult the current MCP MID listing. Some of the most important elements are listed below.

- Requesting task
- Initiating task priority
- Virtual disk/pack address
- Virtual I/O Descriptor
- Relative I/O Descriptor
- Absolute I/O descriptor - set by the CIO instruction (See Appendix B, CIO Instruction.)
- Virtual R/D
- DLP R/D
- R/D status
- Disk or pack ID number
- ID address
- Event structure used for synchronization ofI/O Completion
- Logical exchange number on which I/O is to be fired
- Channel number on which the I/O was initiated or must be initiated
- Fire on indicated channel request indicator
- Hardware type - copied from IOAT or EU table

- IOAT address
- Size of data not transferred
- Time when I/O complete event is signalled
- Link address to next queue in the system IOCB
- I/O characteristics
- I/O Complete handling digit
- Error handling flags
- Error handling mask
- Error recovery routine

I/O MODULE ROUTINES

The I/O Module contains the routines to build Queue Elements and to handle I/O initiation and completion. These routines are summarized as follows.

Build Queue Element Routines

The I/O information provided in the Q1AREA (in the MCP Data Area) is built into the Permanent Queue Element. The address of the Queue Element is then returned.

Rebuild Values in the Queue Element

This routine rebuilds most of the fields of the Queue Element, assuming that only some of the entries are valid.

I/O Fire Routines

An existing I/O Queue Element is initiated. Different entry points reflect the amount of pre-processing that has been done before firing. This routine will not wait for the I/O Complete signal before returning.

Fire I/O and Wait Routines

An existing I/O Queue Element is initiated. Different entry points reflect how much pre-processing has been done before firing. This routine waits for the I/O Complete signal and checks the R/D before exiting.

Temporary I/O Routines

The I/O information is provided in the Q1AREA (in the MCP Data Area) and is fired via a Temporary Queue Element. This routine waits for an I/O Complete signal and checks the R/D before exiting.

Initiate I/O Routine

The Queue Element is locked and the Relative I/O Descriptor is converted to an Absolute I/O Descriptor. The channels on the exchange are then examined to determine if one is available for the I/O. The I/O is then initiated on that channel and is linked as the in-process element for that channel.

If no channel is available, the I/O is linked into the I/O Exchange Queue for that exchange. This linkage is determined by priority.

I/O Complete Handling Routines

When an I/O is initiated by the I/O Module, the Physical I/O Complete processing is performed by the I/O Complete I.R. when the Physical I/O Complete Interrupt occurs.

I/O Complete Independent Runner

The I/O Complete I.R. handles the occurrence of a Physical I/O Complete. It performs any necessary operations to complete the I/O, free the channel, and initiate the next I/O on it. It is idle until a Normal or Error I/O Complete Interrupt occurs.

SECTION 13
TASK CREATION

OVERVIEW

Several actions are required in order for a task to be created. These steps are taken in the interval between when the task execution is requested and when the first instruction in that task can be executed.

The task execution request can be made through an input media, such as the OCS or ODT. That medium's driver sends information about the request to a control command handler, Command I.R.

Command I.R. calls overlays to evaluate the control record, process text, and create the task. Creating the task involves obtaining memory for the task and setting up its environment. The task is then initialized by the Kernel.

A Global Module routine called Beginning of Task (BOTSK) schedules the task and then obtains the task's Code Area from the Resource Manager.

The task next has to finish Beginning of Job processing. The Memory Manager is called to create a Hyper Call Frame for the task, the return address of which is the address of the task's first instruction. Control is passed back to BOTSK.

Finally, a Return instruction is executed and the return is made to the address of the task's first instruction. Therefore, the next instruction that will be executed will be the task's first instruction.

INPUT MEDIA

A task execution request can be made from several input media: OCS/ODT, Card Reader, Pseudo Card Reader, the user program, the SPO Card, the Program Call, and the MCP. (The Remote SPO has been deimplemented as an input medium for task execution requests.)

Each input media has a value that is stored in the MCP Data Area variable CTL-INF. CTL-INF specifies which input media the task execution request is being made from. CTL-INF values for the input medium are shown in Table 13-1.

Table 13-1. CTL-INF Values for Input Media

INPUT MEDIA	CTL-INF VALUE
MCP	0
Card Reader	2
Program (ZIP)	3
Pseudo Card Reader	6
SPO Card	7
OCS/ODT	8
Program Call	9

Each input medium has the job of passing text along with a control structure, called PAG-INF, to the Command I.R. or to the task created by the request. PAG-INF contains, among other information, security information and the input medium's CTL-INF value.

Command I.R. is the driver that handles the control commands. The method by which the text and PAG-INF are sent to the Command I.R. varies slightly, depending on the particular medium. Each medium is discussed in the following sections.

OCS/ODT Input

The OCS/ODT is driven by the OCS/ODT Independent Runner (I.R.), which waits for input from the OCS or the ODT. Once the OCS/ODT I.R. receives some transmitted text, it calls an MCP overlay called KBD-IN.

KBD-IN scans the text for control information and then sends the text along with the control structure, PAG-INF, to the Command I.R.

KBD-IN then waits for Command I.R. or the task to send it a talk message.

Program and Program Call Input

There are four ways for a user program to request the execution of another job:

- ZIP BCT,
- ZIP SPO BCT,
- Program Call BCT, and
- SORT. BCT.

In the case of the ZIP BCT and the ZIP SPO BCT, the MCP function associated with the BCT makes a call to KBD-IN. KBD-IN scans the text for control information and passes it along with the PAG-INF control structure to the Command I.R.

In the case of the Program Call BCT and the SORT. BCT, the MCP function associated with the BCT creates the task and sends the text and PAG-INF to the task. It then initiates the task by issuing an interrupt (INT instruction) to the Kernel and waits for a response from the SORT. task or the Program Called task indicating whether or not the task got to BOJ successfully. (See Section 7, Initiate Task Request).

MCP Input

The MCP may request task execution. Examples include situations such as when:

- the MCP builds control text in order to initiate the RJE Control Task,
- the MCP builds control text and creates the task that initiates the SORT. intrinsic in response to a user's SORT BCT, or
- the MCP creates the task for execution of a Data Base Program (DBP).

SPO Card Input

A SPO card can be contained within a program's data deck. After a SPO card is detected while reading program data cards, the CTL-INF value is set to 7, and KBD-IN is called to process the SPO request.

Card Reader Input (CR)

The Status Module reads the first card and sends the card text and the PAG-INF control structure to the Command I.R.

Pseudo Card Reader Input (PCR)

An example of a task execution request made through the PCR media might be a Compile or an Execute control card in a PCR file that has been created from the Load Control (LDCNTL) program, the Edit/Copy (EDTCPY) program, or some other program that builds a control deck.

If the Automatic PCR (APCR) option is set, then the file will be activated automatically. If the APCR option is not set, then the PCR must be activated by the RN command.

The RN command causes an MCP overlay associated with the RN command processing to open the PCR file and read its first record. The first card in a PCR file must be a control card. The overlay sends the text along with the PAG-INF structure to the Command I.R.

COMMAND I.R.

Command I.R. is the driver that handles control commands. It remains at a Receive entry point while it is not processing any text, and waits for input from one of the input media drivers.

Command I.R. receives two items of information as input from the input media driver:

- text and
- the PAG-INF control structure.

If task execution is requested through the CR or the PCR, then the communication process between Command I.R. and the message sender is asynchronous. That is, the sender does not wait for a talk message from Command I.R.

In contrast to asynchronous communication, which occurs between Command I.R. and either the CR or the PCR, the communication between Command I.R. and the other input media drivers is synchronous. Thus, if task execution is requested through the OCS/ODT, KBD-IN waits for a talk message from Command I.R.

When one program ZIPs another program, no result is returned to the originator. However, when a ZIP SPO is issued, the original program is informed of certain execution results of the ZIPped program, such as its Run Log ID Number.

After Command I.R. receives the text and PAG-INF from the input media driver, it calls an entry point in the Control Card Overlay (CTLCD). This entry point is termed Control Card Evaluate (CC-EVL). CC-EVL initiates processing of the control text as described below.

The control card is evaluated, and the Control Card Table is searched for valid keys corresponding to the given command. The Control Card Table contains a 12-digit entry for each key. The entry contains command priority information, a flag indicating whether or not task creation is required for that command, and the environment and address of the required procedure to process the command.

Text processing continues until one of the following three conditions arises:

- an error is detected,
- a task is created, or
- all of the text belonging to the control command is exhausted.

The Case Where an Error is Detected

An error is signified by a non-zero value in the error code variable CTL-ERR. If an error occurred and the CTL-INF value indicates either CR or PCR media input, then an error message is displayed. Command I.R. then goes back to Receive Mode.

If CTL-INF indicates an input media other than the CR or PCR, then Command I.R. talks to the input media driver in order to give KBD-IN the CTL-ERR value and release the driver.

The Case Where a Task is Created

If there were no errors and a task was created while CC-EVL was running, then an interrupt to the Kernel is issued in order to initialize the task. Command I.R. then returns to Receive Mode.

The Case Where All Text Belonging to the Control Command is Exhausted

If there were no errors and no task was created, Command I.R. calls a Read Text (RDTEXT) routine in order to read another card if the CTL-INF value indicates an input media of either the CR or the PCR.

If CTL-INF indicates an input media other than the CR or PCR, then RDTEXT will set End of Text. This will cause Command I.R. to go back to Receive Mode instead of calling CC-EVL again to process the new card read by RDTEXT.

Return to Command I.R. from CTLCD

Thus far, we have seen that Command I.R. is returned to in the following cases:

- an error is detected,
- a control key is processed that creates a new task, or
- the End of Text/Card is detected.

CREATION OF THE TASK

As discussed in the previous section, Command I.R. calls the CTLCD overlay in order to process text. Once CTLCD finds a task initiate command such as Execute or Compile, it must call the Memory Manager's Create Task routine in order to create the task.

Create Task must do several things in order to create the task, including:

- set up the task's environment by allocating its Environment and Memory Area Tables,
- define the Task State,
- create a Reinstate List entry for the task, and
- assign a Task Number.

Create Task can encounter any of the following three error conditions while it is running.

Enough memory may not be available in order to create the Data Area.

The Maximum Number of Jobs limit can be exceeded.

More than 15 Data Base Programs can be on the system.

If an error does occur, its value is recorded in CTL-ERR.

Create Task returns two values to Command I.R.: the Task Number and a flag indicating whether or not task creation was successful.

If a simple control card was processed, with no task creation involved, it is necessary to return directly to the input driver in order to release the OCS/ODT I.R. or the program initiating the request.

The CTL-ERR value, which was set in the event of an error during control card processing, is passed to KBD-IN. Thus, KBD-IN can check for control errors after it is returned to from Command I.R. The Keyboard Out Module is called in order to display the error, if there was one. The caller is then returned to the program or the OCS/ODT driver.

If a task was created, the rest of the text must be processed by the task itself. Command I.R. resends the control text and the PAG-INF structure to the task and initiates the task through a Kernel request.

BEGINNING OF TASK PROCESSING (BOTSK)

BOTSK is the Global Module routine that the task begins executing after it is created by the Memory Manager and initiated by the Kernel.

BOTSK initiates and completes scheduling of the task, and finally takes the task to Beginning of Job (BOJ).

Scheduling

The scheduling process includes the following actions.

- All of the task's control records are processed.
- The Mix Table entry is set up.
- The Security Module is called to obtain a User Table entry.
- The task is linked to the Mix Table's Schedule Complete List. This action is called the Complete Schedule process.

If a delimiter record (such as an ?EXECUTE, ?COMPILE, or ?END record) is detected during the processing of control records for a task, then the Complete Schedule routine is called and the remainder of the text is resent to Commmand I.R.

The Mix Table contains two lists: the Schedule Complete List and the Running Mix List.

The Schedule Complete List contains an entry for each task for which all control cards have been processed, but for which Code Area has not yet been allocated. The Running Mix List contains an entry for each task for which both Data and Code Areas have been defined.

If any error is detected during the scheduling process, then the Terminate Scheduling Task (TRMSTK) routine in the Global Module is called to terminate the scheduled task.

After scheduling has been completed, an entry is made in the Program Name Table indicating that the task has been scheduled. It is then necessary to obtain the task's Code Area.

Allocation of the Task's Code Area

If, after scheduling has been completed for the task, an End of Text is detected and any handler is waiting to be released, then the task talks to the handler. The task passes its Run Log ID to the handler, waits for a reply from the handler, and then continues.

A message is sent to the Resource Manager I.R. requesting memory for the task's Code Area. Then BOTSK calls the Waiting Memory (WTGMEM) routine in the Waiting Module and waits for a Memory Available message from the Resource Manager.

When the Resource Manager receives the memory request message for the task, it calls the Memory Manager for the requested task's memory. If the memory is available, the Resource Manager will wake up the task with a message indicating that memory was available.

If the Memory Manager call fails, then the task is added to the Resource Manager's Waiting Schedule Memory Table until memory becomes available. When memory is available, the task is awakened.

BEGINNING OF JOB PROCESSING (BOJ)

Once the task has both its Data Area and Code Area, it is almost ready to run. BOTSK calls the BOJ overlay to complete the BOJ processing for the task. BOJ performs the following actions.

- First, the task is taken off of the Schedule Complete List and entered onto the Running Mix List of the Mix Table.
- The Program Parameter Block (PPB) is read and the Program Segment Dictionary (PSD) is built.
- The Program Global is read into the Code Area.
- The Memory Manager is called to set up the Task State. This is accomplished by creating a Hyper Call Frame for the task, as if the user task had executed a Hyper Call to the MCP. The Hyper Call return address is the address of the first instruction of the user program.
- The Program Name Table entry is updated to show that the task is running.
- If there are any Insert or Value parameters, they are loaded into the program.
- A Run Log record is created, and the BOJ message is displayed.

At this point, we are returned back to BOTSK. BOTSK then makes a Kernel request to update the Task Processor Priority field of the task's Reinstate List entry.

Finally, depending on whether or not the task was either a SORT. task or a Program Called task, one of the two following steps is taken in order to at last begin execution of the new task.

- In the case of a SORT. BCT or Program Call BCT, a conversation is initiated to the calling program in order to inform it of the called task's successful BOJ. When the calling program receives the reply, it executes the Return instruction (OP=63). In all other cases, BOTSK executes a Return instruction (OP=63).

The next instruction that will be executed is the first instruction in the user program.

APPENDIX A
DEBUG FACILITY

GENERAL

The MCP Debug Facility is an interactive menu-driven tool for debugging the MCP and user programs at the machine language level. A maximum number of ten tasks can be included in a single session on a V Series machine.

All Debug sessions are initiated by entering commands at the ODT. The ODT subsequently becomes an interactive terminal for that session. The user can switch back and forth between sessions and the ODT by entering the appropriate commands.

The following sections cover in detail the Debug menus, their functionality, and their commands.

SYSTEM REQUIREMENTS

The required hardware is listed below.

- a V Series machine
- a V300 PANDMV floppy with the latest Revision F (or greater) control store
- a V300 PANDMV, or possibly PANSMV floppy if the Maintenance Panel is to be used in conjunction with the Debug Facility

COMMANDS TO INITIATE A DEBUG SESSION

The DEBUG, ID, ED, and QD commands can be issued at the ODT in order to initiate a Debug session. Each of these instructions is discussed, along with the associated error messages.

After one of the above commands is entered, the ODT becomes a terminal for a Debug session. It is RESET and internally reprogrammed for the Forms Mode. Following the Debug session, the terminal will be reinstated to return it to its original state. Note that all screens are cleared by this operation.

The DEBUG Instruction

The syntax of the DEBUG instruction is as follows.

DEBUG <program–id> [ON <medium–id>] [<parameter–list>]

The DEBUG instruction is used to execute program on disk or pack. This instruction is to be entered from an ODT and thereafter the ODT will become an interactive device for the session. The user will be able to switch back and forth among any of ten current Debug sessions.

The Interactive Debug Instruction (ID)

The ID instruction syntax is

<task–number>ID

where task–number is any active job number in the mix.

This instruction is used to debug a task in the active mix. A Debug session is then invoked at the ODT at which the ID command was issued. The task corresponding to the task number is then attached to the session.

This instruction is to be used in order to debug a task that is in an indeterminate state of execution such as a deadlock, infinite loop, or during the execution of an Independent Runner.

The Enter Debug Instruction (ED)

The syntax of the ED instruction is

ED[session-number>

where session-number is a unique four digit number that is prominently displayed within each session.

The ED instruction allows the user to go back to a Debug session of his choice from the ODT. The session-number follows the keyword ED because it identifies an Independent Runner that controls the session (as opposed to a normal Task Number). Only sessions created on that particular ODT are valid session-number values.

This instruction is a companion to the ODT command within a Debug session, ID. While the ID command allows the user to freeze the debugged task and return to the ODT, the ED command allows the user to return to any Debug session that was initiated from that ODT.

The Query Debug Instruction (QD)

The syntax of the QD instruction is shown below.

QD

The QD instruction enable a user to query the ODT about the number of active Debug sessions on the system, along with their associated session numbers.

Error Messages

The error messages that the DEBUG, ID, ED, and QD instructions can generate are listed below, along with their meanings.

ERROR	MEANING
EXCEEDED DEBUG SESSION LIMIT	There are currently ten active DEBUG sessions on the system. No new sessions can be invoked until at least one of the current sessions terminates. QD can be used to identify the current sessions.
ERROR IN INITIATING DEBUG SESSION	There has been an internal error in the sessions.
TASK CURRENTLY BELONGS TO ANOTHER SESSION	The task that the user is attempting to attach to is currently being debugged in another session.

ERROR	MEANING
DEBUG SESSION DOES NOT BELONG TO OCS	The session number does not correspond with the ODT that this command was issued on.
DEBUG SESSION UNASSIGNED	The session number is invalid and does correspond to any current session.
FOUR DIGIT SESSION NO REQD	A four digit session number is to be included in the command.

STATE INFORMATION FOR A DEBUGGED TASK

The Debug facility contains several menu screens as well as an on-line help facility: the Main Menu, the Status Menu, the Trace Menu, the Breakpoint Menu, and the State Menu. Each of these menu formats will be covered in depth in following sections.

All of the menus display the state of the debugged task on the second line. The state information includes the following:

- TASK - the task number assigned to the task
- MODE - the Debug mode
- STATE - the internal state of the task:
  presently debugging,
  stopped,
  running, or
  breakpointed
- ENVRMT - the task's current environment
- NIA - the next instruction address to be executed
- COMS - the comparison toggles
- H/A/O/V/T - tells which breakpoints have been set:
  H - Hyper Call / BCT
  A - address
  O - opcode
  V - overflow
  T - taken branch
- T/P/M/D/U - tells which trace commands have been set:
  T - trace
  P - path
  M - monitor
  D - module
  U - user

THE USER INTERFACE MENUS

The Main Menu appears when the task is in a runnable state. If the task is non-runnable, the Status Menu will appear. From the Trace Menu the user can trace and record the execution of the task up to a specified point. From the Breakpoint Menu the user can set breakpoints in the instruction stream where execution is to stop. The State Menu gives the user access to memory for the purpose of examining and/or altering memory contents.

MENU COMMANDS AND THEIR DEFINITIONS

Various commands can be given while the user is in a menu. All of the menu-level commands are defined in Table A-1, and Table A-2 shows which commands are valid for each of the menus.

Table A-1. Debug Menu Command Definitions

COMMAND	DEFINITION
REF	refreshes the screen by redisplaying the current menu
STOP	causes the task to stop the execution of instructions
SI	allows single instruction execution
RUN	allows execution of a specified number of instructions before stopping again The RUN command syntax is: RUN <n> where n is a 4 digit number of instructions.
GO	allows execution until either an error condition arises or a breakpoint is reached
IGN	allows the reported error to be ignored when the task is armed or when the task receives an asynchronous message
QUIT	detaches the task from the session and ends the session Note that QUIT should be used to terminate a session where a system Independent Runner is being debugged, since QUIT ensures that the task will be detached.

Table A-1. Debug Menu Command Definitions (cont)

COMMAND	DEFINITION
END	terminates the session Note that this command should not be used to end the session when debugging a system Independent Runner. Use the QUIT command instead.
PEEK	displays memory at the indicated offset and memory area for the specified length The PEEK command syntax is: PEEK <offset>,<length>,<memory area>, <environment number>,<task number> where the environment and task numbers are optional. The default values are the debugged task's current environment and task numbers. The digit length of the parameters are: offset - 6 digits length - 3 digits memory area - 2 digits environment number - 6 digits task number - 4 digits
POKE	allows the user to change mem- ory by inserting a "P" at the begin- ning of any lines in the display area
RLE	displays the Reinstate List Entry of the specified task number
MODE	changes the Debug mode to any of the following modes:
USR	enable Debug for only the User Program portions of the task
MCP	enables Debug for only the MCP portion of the task

Table A-1. Debug Menu Command Definitions (cont)

COMMAND	DEFINITION
SYS	enables Debug for both the MCP and USER portions of the task
REL	disables Debug completely until either the task terminates or an error is reached For those portions of the task where Debug is disabled, the task will run at the normal processor speed. All commands that resume execution of the task and terminate the task can be entered only when the debugged task is in a stopped state.

Other menus can also be displayed from the Main Menu. These commands are summarized below. All menus except the Help Menu can be entered only when the task is in a stopped state.

HELP	display a series of Help Menus for the Main Menu
TRACE	allows Trace commands to be given
STATE	allows the task's state and data structures in memory to be displayed and/or modified
ODT	allows the user to switch to the ODT. To return back to the session, the ED command must be issued.
BREAKPOINT	allows breakpoints to be set
RET	allows user to return to the Main Menu from any other menu.

Table A-2. Summary of DEBUG Menu Commands

COMMAND		MAIN MENU	STATUS MENU	TRACE MENU	BREAK-POINT MENU	STATE MENU
REF		X	X			
STOP		X				
SI		X				
RUN		X				
GO		X				
IGN		X				
QUIT		X				
END		X	X			
PEEK		X	X			
POKE		X	X			
RLE		X	X			
MODE-	USR	X	X			
	MCP	X	X			
	SYS	X	X			
	REL	X	X			
HELP	X	X	X	X	X	X
TRACE		X				
BREAKPOINT		X				
STATE		X				
ODT		X				
RET			X	X	X	X

THE MAIN MENU

The Main Menu appears only when the task to be debugged is in a runnable state. The task can then be attached to a session.

The task can be initially in a waiting state, such as it is when it is waiting on a resource. In this case, the Status Menu will appear. The Main Menu will appear when the task eventually becomes runnable again.

From the Main Menu, the user can enter the Trace, Breakpoint, and State Menus for the purpose of executing code in an observable, controlled manner.

Judicious use of the Modes (USR, MCP, SYS, or REL) can reduce debugging time significantly. For example, if a user wishes to debug a user program, there is no need to fault after each instruction executed within the MCP code. Use of the USR Mode would prevent this.

THE STATUS MENU

This menu is the primary menu for all tasks that are not runnable.

THE TRACE MENU

This menu is entered from the Main Menu. It allows the user to direct a variety of trace output to three types of medium at the same time.

The types of trace actions are:

TRACE	traces all instructions executed until the task terminates, an error occurs, or a breakpoint is reached.
PATH	produces a trace line when instruction execution ceases to be sequential, as with Taken Branch instructions and Enter
MONITOR	produces output lines whenever the task enters or exits a procedure
MODULE	produces output lines whenever the task enters or exits a procedure and the environment changes
USER	produces output lines whenever the task executes a user-selected opcode
USER SELECTED OPCODE	allows the user to specify a maximum of ten 2-hex-character opcode

Any combination of the trace actions can be output to any device. The output options are:

ODT	Enter any non-blank character
PRINTER	Enter <channel number>/<unit number> where channel number is a 4 digit number and unit number is a 2 digit number
PBD or PBT	Enter any non-blank character for Printer Backup Disk or Printer Backup Tape Only one Printer Backup File can be opened at a time.

THE BREAKPOINT MENU

This menu enables the user to set a maximum of 32 breakpoints on 10 Hyper Call/BCT instructions, 10 instruction addresses, 10 opcodes, or an overflow or taken branch. All breakpoints can be easily set or reset by entering any non-blank character in the able/disable field that proceeds each breakpoint. The commands for setting breakpoints are discussed below.

The Hyper Call/BCT Breakpoints

The commands for setting this breakpoint are shown below.

COMMAND	DEFAULT VALUE
Hyper Call / BCT NUMBER <4 digit number>	<0000>
OFFSET <2 digit number>	<00>
MASK <6 hex value>	<F0F0F0>
COUNT <2 digit number>	<00>

The offset is the number of digits relative to the start of the HCL / BCT parameters.

The mask is compared with the offset value and the task is stopped on a prefect match.

The count specifies the umber of occurrences of the Hyper Call / BCT to be skipped before the task is brought to a halt.

The Address Breakpoint

The commands necessary for setting this breakpoint are listed below.

COMMAND	DEFAULT VALUE
ADDRESS OFFSET <6 digit number>	<000000>
ENVIRONMENT NUMBER <6 digit number>	<000000>
COUNT <2 digit number>	<00>

The address offset is the instruction address the task is to be stopped on.

The MAT number is the Environment Table number.

The count specifies the number of occurrences of the address that are to be skipped before the task will stop.

The Opcode Breakpoint

This breakpoint stops execution at a given number of occurrences of a specified instruction. The commands for setting this breakpoint are below.

COMMAND	DEFAULT VALUE
SYMBOLIC OPCODE <3 character string>	< >
AFBF <4 digit number>	<FFFF>
COUNT <2 digit number>	<00>

The symbolic opcode is the mnemonic for the opcode.

The AFBF field is the actual value that is compared to the code stream before the task is stopped.

The count specifies the umber of occurrences of the opcode ignored before the task is stopped.

The Overflow Breakpoint

The user can stop the task once an executed instruction causes the Overflow Toggle to be set.

The Taken Branch Breakpoint

This breakpoint allows the user to stop the task once the sequential flow of the instruction stream has been broken.

THE STATE MENU

The user can display and modify the task's state and data structure in memory. Table A-3 lists all of the commands for this menu and their function.

Table A-3. State Menu Commands

COMMAND	DEFINITION
STK	displays the Topmost Stack Frame
NSTK	displays the Next Stack Frame relative to the last one displayed
PSTK	displays the Previous Stack Frame relative to the last one displayed
NIA	displays/modifies the Next Instruction Address

Table A-3. State Menu Commands (cont)

COMMAND	DEFINITION
COMS	displays/modifies the task's Condition Toggles The encodings are as follows: (2) O - Overflow is set E - COMS are set EQUAL N - COMS are set NULL (1) L - COMS are set LOW (0) H - COMS are set HIGH
INT	displays/modifies the Interrupt Mask The encodings are as follow: (5) O - Processor over temperature (4) T - Timer (2) R - Real-Time I/O (1) E - Error I/O O - No bits set (0) N - Normal I/O
MOD	displays/modifies the Mode Toggles The encodings are as follows: (8) F - Soft Fault enabled (4) P - Privileged Task O - No toggles set (2) T - Trace enabled (0) S - Snap enabled
MOP	displays/modifies the Measurement Register
ACC	displays/modifies the current Accumulator
IXn	displays/modifies one of the processor's Index Registers "n" is a number between 1 and 7.

DEBUG SESSION ERRORS

When the processor is executing on behalf of a debugged task and encounters an error, it reports the error to the session through a Hardware Interrupt. The exact nature of the fault can be indicated in the Fault Indicators.

Table A-4 lists the additional information that the Invalid Instruction and Address Error Faults generate.

Table A-4. Fault Codes Generated by the Invalid Instruction and Address Error Faults

DIGIT	BIT	TYPE OF FAULT
72	0	Soft Memory Area
72	1	Invalid Arithmetic Data
72	2	Trace
72	3	Hard Memory Area
73	0	Instruction Timeout
73	1	Address Error - More information in the AEX Byte
73	2	Uncorrectable Memory Parity Error
73	3	Invalid Instruction - More information in the IEX Byte
74	0	Soft Fault
74	1	Snap Picture taken
74	2	Accumulator Trap taken
74	3	Stack Overflow

APPENDIX B
V SERIES EXPANDED INSTRUCTION SET

GENERAL

Several new instructions have been created for the V Series. Some instructions are user instructions; others are privileged instructions and can only be used by the operating system. They are briefly discussed below, and thorough documentation of the individual instructions is on the following pages.

Alter Environment Entry (AEE)

This powerful and potentially dangerous privileged instruction is used to alter an ET entry. It is only used by the operating system. It is a variant of the ATE instruction.

Alter Table Entry (ATE)

This privileged instruction changes a MAT entry Copy Descriptor (one of the #0 through #7 entries, for example) to point to another entry (one of the #88 through #99 entries, for example). The updated MAT is then reloaded.

Write Hardware Register (WHR)

WHR is a privileged instruction used by the MCP to establish the absolute memory addresses of the Reinstate List, Snap Picture, Memory Error Report, or Memory Area Status Table.

Virtual Enter (VEN)

VEN allows a user to change addressing environments. The first operand tells the processor where the parameters are (in a data memory area). The second operand specifies the environment to enter. The instruction then loads the new MAT.

There are local and non-local VENs. A local VEN does not require that the new MAT be loaded, while a non-local VEN does require switching to a new MAT and loading it.

Branch Reinstate Virtual (BRV)

BRV is a privileged instruction which is used to dispatch a task.

Hyper Call (HCL)

This instruction is used by a task to call an MCP function. HCL will eventually replace the BCT instruction. The first operand points to the parameters and the second operand described the environment to be entered.

Return (RET)

This generic Return is used for returning from Hyper Call, BCT, and VEN. The processor first checks the stack frame and pops the stack accordingly to return.

Interrupt (INT)

INT is a privileged instruction which causes a transfer of control to the MCP Kernel.

Initialize Lock/Event Structures (ILS)

ILS is a privileged instruction which creates and initializes a Lock Structure or an Event Structure in memory, or performs a counted wait on an Event.

Move Lock Structures (MLS)

MLS is a privileged instruction which moves a Lock Owner from a Lock Structure, or an Event Count from an Event Structure, to a destination field.

Lock (LOK)

The LOK instruction allows the operating system to preserve the integrity of all shared data structures. LOK locks other users out while a data structure is being accessed by a task.

Load Index Register (LIX) and Store Index Register (SIX)

These instructions allow the user to manipulate the Mobile Index registers (IX4, IX5, IX6, and IX7), as well as IX1, IX2, and IX3.

Search Table (STB) and Search List (SLT)

Both STB and SLT allow a user to search with a key of greater than 100 bytes. The lists or table elements may be of greater size than 100 bytes.

These instructions also allow the user to search memory areas outside of the user's data or code (Memory Area 0 and Memory Area 1, respectively).

Move String (MVS) and Compare String (CPS)

These instructions allow the movement or comparison of large pieces of data. Data can be moved from or compared to different memory areas.

Hash String (HSH)

The HSH instruction contains a hashing algorithm and returns a hash number.

Adjust Stack Pointer (ASP)

One stack is allocated in the operating system for each user. If these are local stack variables, ASP increments the Top of Stack (TOS) pointer.

It then follows an algorithm to make certain that there is enough stack available to call the Fault Handler in case an error occurs or if stack space begins to run out. (VEN also contains this algorithm.)

If there is not enough stack, ASP causes a Stack Overflow Fault and invokes the Fault Handler. The Fault Handler invokes the Stack Overflow I.R. which performs the stack expansion.

Convert I/O (CIO)

CIO converts a relative address (of a buffer or an I/O Descriptor) to an absolute address for the IOP. This absolute address is returned to the MCP.

To prevent the Memory Manager from moving memory areas between the time of the calculation completion and the execution of an Initiate I/O instruction, CIO sets a flag in the MAST. This flag indicates to the Memory Manager that this memory area is not to be moved until the I/Os in Process Count equals zero.

CIO is a privileged instruction.

I/O Complete (IOC)

This privileged instruction tells how much data was moved on the I/O. It also decrements the I/Os in Process Count and resets the flag.

System Status (SST)

This privileged instruction gives the status of the machine including memory area reports, Snap Picture, over-temperature, processor type, number, system number, serial number.

Measurement OP (MOP)

The MOP instruction is used to monitor addresses, procedures, or other elements of interest. MOP is followed by a bit pattern which is sent to one the backplanes when it is encountered by the processor. A hardware monitor connected to the backplane pins can then be used to record the number of times different MOPs are executed.

Fail (BAD)

The Fail instruction causes an intentional Invalid Instruction fault (IEX=01). Its primary use is in software debugging.

ADJUST STACK POINTER (ASP) OP = 61

Function

The Adjust Stack Pointer instruction is used to increment the value of the Top of Stack Pointer, provided that there is sufficient space between the pointer and Limit #0. (The top of Stack Pointer is stored at address 40 of the Kernel Data Area.)

Format

61	AF	BF	A

AF Field

A length of six, "6", must be specified in the AF field directly, as an indirect field length, or as a literal.

BF Field

The BF field is unused and reserved. It may be specified as indirect.

A Field

The A field contains the address of the increment field. The address may be indexed, indirect, or extended. The final address controller must be UN or an Invalid Instruction fault will be caused (IEX=03).

Implementation

The sum of the increment value (A), the Top of Stack Pointer value, the value of Base #0, and the size of the Hardware Call Stack Frame area (500) is then compared to Limit #0.

If the sum is greater than or equal to the Limit #0, then a Stack Overflow fault occurs which stores the address of the failing instruction. The instruction is terminated and no further action is taken.

If the sum is less than Limit #0, then the sum of the Top of Stack Pointer value and the increment value (A) is stored into the Top of Stack Pointer.

ALTER TABLE ENTRY (ATE) OP = 86

Function

This instruction is used to alter an Environment Table or Memory Area Table entry, and may only be executed in the Privileged Mode.

Format

86	AF	BF		A	B

AF Field

The AF field is unused and reserved. It may be specified as an indirect field length. A literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

The BF field is a variant and may be specified as an indirect field length. Any BF values other than those defined below will cause an Invalid Instruction fault (IEX=26).

BF	=	00	marks the selected Memory Area Table entry as unused.
BF	=	01	performs a copy of a Memory Area Table entry.
BF	=	02	alters the task's Environment Table entry.
BF	=	03	alters the task's Memory Area Table entry.
BF	=	04	signals to the processor that a Memory Area Table entry was modified by an instruction other than ATE.

A Field

The A field contains the address of the source operand. This address may be indexed, indirect, or extended. The final address controller must equal UN or an Invalid Instruction fault will be caused (IEX=03).

When BF = 00 or BF = 04, the A operand is unused. However, this address must still have valid address syllable attributes.

B Field

The B field contains the address of the destination operand, and may be indexed, indirect, or extended. The final address must equal UN or an Invalid Instruction fault will be caused (IEX=03). When BF = 04, this operand is unused but the address must still have valid address syllable attributes.

Implementation

The five functions corresponding to the five valid variant values are discussed below.

The Case Where BF = 00 (WRITE UNUSED MAT ENTRY)

The Write Unused Memory Area Table (MAT) entry variant is used to store an Unused entry in the MAT entry specified by the Environment Number (EN) and the Memory Area Number (MAN) found in the destination operand (B).

The 8-digit destination operand (B) contains the following necessary information for locating the desired MAT entry.

INFORMATION	DIGITS
Environment Number	00 - 05
Memory Area Number	06 - 07

If the Destination Write Enable bit in the ET entry associated with the destination operand is set, the destination entry is located with the EN and the MAN found in B, as above. Otherwise, an Invalid Instruction fault (IEX=36) is caused and no further action is taken.

If the Type digit of the destination entry is an Original entry or a Memory Area Fault entry, an Invalid Instruction fault is caused (IEX=36), the instruction is terminated, and no further action is taken.

If the Type digit of the destination entry is an Original entry or a Memory Area Fault entry, an Invalid Instruction fault is caused (IEX=35), the instruction is terminated, and no further action is taken. Otherwise, an Unused entry is written in the specified destination.

Finally, the current MAT is reloaded using the Active Environment Number.

The Case Where BF = 01 (COPY MAT ENTRY)

The Copy MAT entry variant is used to generate a copy of the source MAT entry specified by the EN and the MAN found in the source operand (A). This copy is then stored in the destination entry location specified by the EN and the MAN found in the destination operand (B).

Each 8-digit operand contains the following information necessary for locating the desired MAT entry.

INFORMATION	DIGITS
Environment Number	00 - 05
Memory Area Number	06 - 07

The operations performed by this variant are discussed below.

First, the destination MAT entry must be resolved from the EN and the MAN in the destination operand (B). If the Type digit of this resolved destination entry is an Original or a Memory Area Fault entry, an Invalid Instruction fault (IEX=35) is caused, the instruction is terminated, and no further action is taken.

Second, the source MAT entry must be resolved from the EN and the MAN found in the source operand (A). If either of the following bits is not set, an Invalid Instruction fault (IEX=36) is caused:

- the Source Copy Enable bit in the ET entry associated with the source operand (A), or
- the Destination Write Enable bit in the ET entry associated with the destination operand (B).

If the resolved source MAT entry belongs to the current task, then a "C" Copy Descriptor will be stored which points to that resolved source MAT entry.

If the resolved source MAT entry belongs to another task, then an "E" Copy Descriptor will be stored which points to that resolved source MAT entry.

Finally, the current MAT will be reloaded, using the Active Environment Number.

The Case Where BF = 02 (ALTER TASK'S ET ENTRY)

This variant is used for altering an existing ET entry which is specified by the destination operand (B). This entry is to become a copy of the ET entry specified by the source operand (A).

The source operand (A) format is:

INFORMATION	OFFSET
Environment Number	00 - 05
Task Number	06 - 09

The source EN must be a valid MCP or USER EN. The source Task Number must be a decimal value.

The destination operand (B) format is:

INFORMATION	OFFSET
Environment Number	00 - 05

The destination EN must be a decimal non-zero value.

The operations performed by this variant are discussed below.

First, the source Reinstate List entry is located by using the Task Number as an array subscript into the Reinstate List.

Second, the source ET is located from the ET address contained within the source Reinstate List entry. The EN in the source operand (A) is used as an array subscript into this ET. If the source EN is greater than the Reinstate List's Number of Entries in ET field value, an Address Error fault occurs (AEX=57), the instruction is terminated, and no further action is taken.

Third, the destination entry is located by using the EN (contained in the B operand) as an array subscript into the appropriate ET.

If the first digit of the EN is equal to "0 - 9", then this 6-digit number represents an array subscript into the current User ET of 000000 - 999999. If the EN is equal to or greater than the value of the Number of Entries value in the User ET field (located in the Reinstate List entry for this task), then an Address Error fault (AEX=57) is caused, the instruction is terminated, and no further action will be taken.

If the first digit of the ET is not "0 - 9", an address Error fault (AEX=52) will occur and the instruction will be terminated with no further action.

Finally, the contents of the source ET entry are copied to the destination ET entry. The current MAT is then reloaded, using the Active Environment Number.

NOTE

Two guidelines must be followed with this variant:1. the destination EN must never be equal to the currently active EN, and2. the source MAT (pointed to by the EN/Task Number pair) can never contain Copy Descriptors because they will not be resolved correctly. Any pointers that may be affected by this instruction must be recalculated.

The Case Where BF = 03 (COPY ALTERNATE TASK's MAT ENTRY)

This variant is used to generate a copy of the source MAT entry (specified by the source operand A) and store it into the current task's destination operand B.

The source operand (A) format is:

INFORMATION	OFFSET
Environment Number	00 - 05
Memory Area Number	06 - 07
Task Number	08 - 11

The destination operand (B) format is identical to that of the BF = 01 variant case:

INFORMATION	OFFSET
Environment Number	00 - 05
Memory Area Number	06 - 07

The operations performed by this variant are described below.

First, the destination MAT entry is located and resolved, using the EN and the MAN from the destination operand B.

If the Type digit of the destination entry is an Original entry or a Memory Area Fault entry, an Invalid Instruction fault (IEX=35) will occur, the instruction will be terminated with no further action.

The source MAT entry is located and resolved, using the Task Number, EN, and MAN contained in the source operand A.

If either of the following two bits are not set, an Invalid Instruction fault (IEX=36) will occur, and the instruction will be terminated with no further action taken:

- the source Copy Enable bit in the ET entry associated with the resolved source entry, or

- the destination Write Enable bit in the ET entry associated with the destination operand.

Next, an "E" Copy type entry, containing the absolute address of the resolved source entry A, is stored into the destination entry B.

Finally, the Active Environment Number is used to reload the current MAT from memory.

The Case Where BF = 04 (NOTIFICATION OF MAT MODIFICATION BY AN INSTRUCTION OTHER THAN ATE)

This variant performs no significant operation other than to notify the processor that a MAT entry was changed by an instruction other than ATE.

It is necessary for those processors which cache Memory Area Base/Limit pairs to always be notified by some type of ATE instruction when a MAT entry is modified.

FAIL (BAD) OP = AB

Function

This instruction causes an intentional Invalid Instruction fault (IEX=01).

Format

AB AF BF A

AF Field

The AF field is unused and ignored. The literal and indirect field length flags are ignored.

BF Field

The BF field is unused and ignored. The indirect field length flag is ignored.

A Field

The A field is unused and ignored. The indirect, extended, and indexed flags are ignored.

Implementation

The Fail instruction causes an intentional Invalid Instruction fault (IEX=01). The instruction is not examined for any invalid address constraints.

VIRTUAL BRANCH REINSTATE (BRV) OP = 93

Function

This instruction may only be executed in Privileged Mode and is used to dispatch a task. It is a companion instruction to the Interrupt procedure because it transfers control from the MCP Kernel environment to a user task. The processor's registers are restored according to the contents stored in the Interrupt Frame, and control is transferred to the task specified by the address of the Reinstate List entry pointer stored in IX1. (See Section 9, Interrupt Frame.)

Format

93 AF

AF Field

The AF field is unused and reserved.

Implementation

The operations performed in order to exit the MCP Kernel environment and enter the user task environment are summarized below.

First, the new task is located in the following way. The IX1 register contents are examined in order to determine the Reinstate List entry for the new task to be executed. From this point forward, this new task is referred to as the Current Task. The Task Number from this Reinstate List entry is stored into the Current Task Number field of the Kernel Data Area (absolute memory addresses 82-85). Kernel Mode is then Reset.

Next, the Current Task's execution status information is brought in. This is accomplished when the Interrupt Frame is loaded from the Reinstate List entry for the task.

Timer related functions are then performed. The Task Time is set to the value of the task's Reinstate List Time Slice Remaining field.

The MOPOK line is set to "zero" while the Measurement register is being changed, and is set to "one" at all other times.

At this point, any number of errors could have occurred, causing BRV to fail. If BRV does fail, the following information will be saved, and then an Instruction Interrupt will be issued to the MCP Kernel.

- Fault indicators are stored into the Failed Hardware Call R/D Area field of the task's RL entry.
- The value "07" is stored into the State Indicator field of the task's RL entry. ("07" is the State Indicator code for "Invalid BRV".)
- The value "07" is stored into the Instruction Interrupt Cause Descriptor field of the Kernel Data Area. ("07" is the Instruction Interrupt Cause Descriptor code for "Failed BRV".)

If an error has not occurred, processing will continue as follows.

The appropriate MAT is then identified and loaded. This MAT is the one pointed to by the Active Environment field value within the Interrupt Frame (mentioned above).

Fault conditions must then be tested for. If the Soft Fault and Trace Fault Condition Indicators warrant it, a Hardware Call Procedure will be executed. (If Trace mode is set, then a Trace Fault condition exists.)

If Soft Fault is enabled, the Soft Fault Pending Flag in the task's Reinstate List entry must be examined. If it is not equal to zero then a Soft Fault Condition exists.

If a Fault Condition has been found, a Hardware Call Procedure must be executed. HCP will store the address of the next instruction to be executed, and report the existing Fault Conditions.

Finally, if no Fault Conditions exist, the new Base/Limit information is used to resolve the next instruction address (relative to Base #1) and an unconditional branch to that address is executed.

CONVERT I/O (CIO) OP = 85

Function

This privileged instruction converts the relative addresses in the initial descriptor to absolute addresses in the resultant descriptor. It then verifies that an I/O can be initiated to the specified memory area. Finally, it increments the Number of I/Os in Process field for that memory area in the Memory Area Status Table (MAST).

Format

85	AF	BF	A	B

AF Field

The AF field is unused and reserved. AF maybe specified as an indirect field length. A literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

The BF field is unused and reserved, but may be used as an indirect field length.

A Field

The A field contains the address of the initial descriptor.

B Field

The B field contains the address of the resultant descriptor.

Both the A and B address may be indexed indirect or extended. The final address controller must specify UN or an Invalid Instruction (IEX=03) will be caused.

The formats of the initial descriptor and resultant descriptor fields follow.

INITIAL DESCRIPTOR FORMAT

INFORMATION	DIGITS
Opcode Syllable	00-05
Environment Number	06-11
Memory Area Number	12-13
A Address (Convert)	14-19
B Address (Convert)	20-25
C Field	26-33

RESULTANT DESCRIPTOR FORMAT

INFORMATION	DIGITS
Opcode Syllable	00-05
A Address	06-15
B Address	16-25
C Field	26-33
Memory Area Status Number	34-39

Implementation

The initial descriptor, A, is read from memory. If the A and B addresses are not MOD 2, if they contain undigits, or if the A address is not less than the B address, then an Address Error fault (AEX=01) is caused, the instruction is terminated, and no further action is taken.

The MAT entry pointed to by the EN and MAN in the initial descriptor A is located and resolved. If the resolved MAT entry is a Memory Area Fault entry, then a Hard Memory Area fault is caused and the instruction is terminated. Otherwise, if the resolved MAT entry is not an Original entry, then an Address Error fault is caused (AEX=04) and the instruction is terminated.

The base value in the resolved MAT entry is added to the initial descriptor addresses A and B. If the resultant addresses are greater than the associated limit value, then an Address Error fault (AEX=24) is caused, the instruction is terminated, and no further action is taken.

The initial descriptor's Opcode Syllable and C field are moved to the the resultant descriptor, as is the MAST Number from the resolved MAT entry. This MAST Number is used as an array subscript into the MAST to locate the MAST entry associated with this memory area. If the Inhibited I/O flag of of this MAST entry is set, then the Comparison Flags will be set to LOW and the instruction is terminated with no further action.

If the Inhibited I/O flag is reset, then the MAST entry's Number-of-I/Os-in Progress field is incremented. If the result overflows the field, then an Invalid Instruction fault (IEX=05) is caused, and the instruction is terminated with no further action.

If there is no overflow, then the incremented value is stored back into the field and the Comparison Flags are set to EQUAL.

COMPARE STRING (CPS) OP = A1

Function

This instruction compares the binary values of one substring to those of a second substring.

Format

A1	AF	BF	A	B

AF Field

The AF field is the A Memory Area variant. AF may be indirect, but a literal flag causes an Invalid Instruction fault (IEX=21).

The most significant digit is the Memory Area variant.

- A value of "0" indicates that the Memory Area specified by the Environment Number and the Memory Area Number contained in the first String Descriptor A is to be used for the addresses contained within the first String Descriptor, A.
- A value of "4" indicates that the Memory Area specified for the first String Descriptor A is to be used for the addresses contained within the first String Descriptor.
- The use of all other MSD AF values is reserved and causes an Invalid Instruction fault (IEX=25).

BF Field

The BF field is the B Memory Area variant. BF may be indirect.

The most significant digit is the Memory Area variant.

- A value of "0" indicates that the Memory Area specified by the Environment Number and the Memory Area Number contained in the second String Descriptor A is to be used for the addresses contained within the second String Descriptor, A.
- A value of "4" indicates that the Memory Area specified for the second String Descriptor A is to be used for the addresses contained within the second String Descriptor.
- The use of all other MSD AF values is reserved and causes an Invalid Instruction fault (IEX=26).

A Field

The A field contains the address of the first string descriptor. This address may be indirect, indexed, or extended. The final address controller must specify UN or an Invalid Instruction fault (IEX=03) is caused.

B Field

The B field contains the address of the second string descriptor. This address may be indirect, indexed, or extended. The final address controller specifies the padding variant as shown below.

PADDING VARIANT	B-CONTROLLER
Pad with zero	0 (UN)
No padding	1 (SN)
Pad with blank (40)	2 (UA)

IMPLEMENTATION

This instruction compares the binary values of the substring defined by String Descriptor A's string begin and end addresses and String Descriptor B's begin and end addresses. The Comparison Flags are set to EQUAL if the strings are identical, HIGH if the A string is of greater value than the B string, and LOW if the A string is of lesser value than the B string. No comparison is made on data located at the string end addresses.

If a string begin address is greater than a string end address, then an Address Error fault (AEX=01) is caused and the instruction is terminated with no further action.

If the source and destination lengths are equal, the settings are compared and the Comparison Flags are set. If the lengths are unequal, then the comparison will be dependent on the value of the B address controller.

If the B address controller is equal to "1", then the longer string is compared to the shorter string for the length of the shorter string. In this case, there is no padding and comparison with a null string always causes the Comparison Flags to be set to EQUAL.

If the B address controller is equal to "0", then the longer string is compared to the shorter string for the length of the shorter string. The remainder of the longer string is then compared against zeroes. (For comparison with a null string, the non-null string is compared entirely against zeroes.)

If the B address controller is equal to "2", then the longer string is compared to the shorter string for the length of the shorter string. The remainder of the longer string is compared against blank characters (40). (For comparison with a null string, the non-null string is compared entirely against blank characters.)

Null Strings

The comparison of two null strings causes the Comparison Flags to be set to EQUAL.

Overlap

Partially overlapping descriptors produces unspecified results

HYPER CALL (HCL) OP = 62

Function

HCL is used to enter a function in the MCP environment. The Top os Stack (TOS) limit is checked, then the processor registers, machine state, and parameters are stored on the stack of the called environment. Control is then transferred to the specified function.

Format

62	AFBF	A	B

AFBF Field

The AFBF field contains the length, in bytes, of the parameter field. The maximum number of bytes that can be moved is 9,999, and a value of 0000 bytes indicates that no data shall be moved. Indirect field lengths may be specified.

AF literals of B1, B2, or B3 are interpreted as lengths of 1, 2, or 3 characters in the A field, respectively. All other literal values cause an Invalid Instruction fault (IEX=22).

A Field

The A field contains the address of the parameter data field operand. This address may be indexed, indirect, or extended. The final address controller must equal UA or an Invalid Instruction fault (IEX=03) will be caused.

B Field

The B field contains the address of the 4-digit Function Number. This address may be indexed, indirect, or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) will be caused.

Implementation

The 6-digit address of the Hyper Call Function Table is located. (at address 87-92 of the Task MCP Data Area.)

The 4-digit Function Number B is used as an array subscript into the Hyper Call Function Table. If the Function Number is not numeric, an Address Error fault (AEX=34) will occur and the instruction will be terminated. If the resultant address exceeds the 6-digit Hyper Call Function Limit (located at address 94 relative to the MCP Data Area), then an Address Error fault (AEX=02) will occur and the instruction will be terminated.

Each Function entry contains the following information.

INFORMATION	DIGITS
Environment Number (EN)	00 - 05
Next Instruction Address	06 - 11
Protection Field (DD)	12 - 13
Reserved	14 - 15
Interrupt Mask	16 - 17
Mode Indicators	18 - 19

If the Protection field does not equal "DD" then an Invalid Instruction fault (IEX=37) will be caused and the instruction will be terminated.

Next, the EN (contained in the Function entry) must be resolved to point to the selected ET entry. (The Active Environment Number is retained so that it may be stored on the stack.)

Entry #0 of the new environment's MAT must be resolved while maintaining addressability with Base #0 of the current environment.

The Top of Stack (TOS) pointer (located at address 40 relative to the new environment's Base #0) is used as the starting address (also relative to the new environment's Base #0) in storing the Hyper Call Stack Frame.

The sum of the TOS pointer, the size of the Hyper Call Stack Frame, the amount of parameters (AFBF x 2), and the size of the Hardware Call Stack Frame area (500) is compared to Limit #0. If the sum is greater than or equal to Limit #0, then a Stack Overflow fault will be incurred, and the instruction will be terminated. Otherwise, the Hyper Call Stack Frame is stored in the following order.

	INFORMATION	DIGITS
Old TOS —->	Accumulator	00 - 27
	Measurement Register	28 - 35
	Interrupt Mask	36 - 37
	Mobile Index Registers	38 - 69
	Mode Indicators	70 - 71
	COM and OVF Flags	72 - 73
	Active Environment Number	74 - 79
New IX3 —->	Next Instruction Address	80 - 85
	Saved IX3 Value	86 - 93
	Stack Frame Indicator (FE)	94 - 95
	Stack Parameters (0-9,999 bytes)...	
New TOS —->		

If there are parameters, they are moved from a location in memory (A) to the Hyper Call Stack Frame.

The new value of the next available stack location, relative to Base #0, is stored into memory location 40, relative to Base #0.

IX3 is then set in the following way. The two most significant digits are set to the value "C0". The six least significant digits are set to the initial address specified at memory location 40 + 80, relative to Base #0. IX3 will now point to the Next Instruction Address in the Hyper Call Stack Frame.

The machine state is set as follows.

INFORMATION	SETTING
Next Instruction Address	Function Table
Active Environment Number	Function Table
Interrupt Mask	Function Table
Mode Indicators	Function Table
Measurement Register (User)	000000
COM and OVF Flags	RESET

The MOPOK line is set to "zero" while the Measurement Register is being changed and set to "one" at all other times.

If Soft Fault is now enabled, the memory location specified by the Reinstate List entry + 8 must be examined. If it is not equal to zero, a Hardware Call Procedure must be executed in order to store the address of the next instruction to be executed. (See Section 10, Soft Faults.)

The MAT pointed to by the Active Environment Number must be loaded. Using the new Base/Limit information, the next instruction address is resolved, relative to Base #1. Finally, an unconditional branch to that address is executed.

NOTE
The use of the Mobile Index registers or the Accumulator in order to pass parameters is invalid.

HASH STRING (HSH) OP = A2

Function

The HSH instruction contains a hashing algorithm and returns a hash number.

Format

A2	AF	BF	A	B

AF Field

AF is a Memory Area variant. AF may be indirect, but a literal flag causes an Invalid Instruction fault (IEX=21).

A value of "00" indicates that the Memory Area specified by the Environment Number and the Memory Area Number contained in the Source String Descriptor (A) is to be used for the addresses contained within the Source String Descriptor.

A value of "40" indicates that the Memory Area specified for the Source String Descriptor (A) is to be used for the addresses contained within the Source String Descriptor.

The use of all other AF values is reserved and causes an Invalid Instruction fault (IEX=25).

BF Field

BF indicates the length of the B field. A value of "00" is equal to a length of 100 digits. BF may be indirect.

A Field

The A field contains the address of the Source String Descriptor. This address may be indexed, indirect, or extended. The final address controller must specify UN or an Invalid Instruction fault (IEX=03) is caused.

If AF has the value "00", then the Source String Descriptor has the following format.

INFORMATION	DIGITS
Environment Number	00 - 05
Memory Area Number	06 - 07

B Field

The B field contains the address of the destination hash key field. This address may be indexed, indirect, or extended. The final address controller must specify UN or an Invalid Instruction fault (IEX=03) is caused.

Implementation

HSH produces a hash key of BF digits based on the string defined by the String Begin and End addresses specified by A. The hashing algorithm requires that successive BF lengths of the specified string be Exclusive ORed and the result be stored in the memory location specified by the B address.

If the String Begin address is greater than the String End address, an Address Error fault (AEX=01) is caused and the instruction is terminated with no further action.

If the length of the source string is less than the length of the destination field, the source string data is moved to the destination data field and the remaining destination data field is filled with trailing zeroes.

If the length of the source string is equal to the length of the destination field, the source string data is moved to the destination data field and the instruction is terminated with no further action.

INITIALIZE LOCK/EVENT STRUCTURES (ILS) OP = 69

Function

This privileged instruction creates and initializes a Lock Structure or an Event Structure in memory, or performs a counted Wait on an Event.

Format

69	AF	BF	A	B

AF Field

The AF field is the length of the A operand in digits. AF may be specified as indirect or as a literal. Its value must be 1, 4, or 6, or an Invalid Instruction fault (IEX=20) will be caused.

BF Field

The BF field is used as an instruction variant and may be indirect. The legal variants are shown below. All other BF values are reserved and will cause an Invalid Instruction fault (IEX=26) if used.

VARIANT	FUNCTION
02	Counted Wait
01	Create Lock
00	Create Event

A Field

The A field contains the address of the initial state data for the Event/Lock being initialized or used. The address may be indexed, indirect, or extended. The final address controller must be UN or an Invalid Instruction fault (IEX=03) will be caused.

If BF equals "00", this is the address of a Boolean value which determines whether the Event Structure being created will initially be in the Happened state or in the Not Happened state.

If BF equals "01", this is the address of the canonical lock number for the created Lock Structure.

If BF equals "02", this is the address of the count value to be used to determine whether or not to Wait.

B Field

The B field contains the address of the Lock/Event Structure being initialized. The address may be indexed, indirect, or extended. The final address controller must be UN or an Invalid Instruction fault (IEX=03) will be caused.

If BF equals "00" or "02", this is the address of an Event Structure.

If BF equals "01", this is the address of a Lock Structure.

Implementation

The Case Where BF=00 (CREATE EVENT)

This variant creates an Event Structure in either the Happened state or the Not Happened state.

If the A operand is non-zero (Boolean true), an Event Structure is created at address B in the Happened state. If the A operand is zero (Boolean false), an Event is created in the Not Happened state.

All other fields in the Event are cleared to zeroes and the Comparison Flags remain unchanged.

The Case Where BF=01 (CREATE LOCK)

This variant creates a Lock Structure in the processor-dependent available state with the Lock Number field set to the value provided by the A operand.

All other fields are cleared to zero and the Comparison Flags remain unchanged.

If the A operand equals zero or contains undigits, then an Invalid Instruction fault (IEX=06 or IEX=07, respectively) is caused and no further action is taken.

The Case Where BF=02 (COUNTED WAIT)

This variant uses the count provided by the A operand as a guard to determine whether or not to perform the Wait function on the Event Structure described in the B field.

The Event Structure is read from memory. If the Event Designator field is not equal to zero, then an Invalid Instruction fault (IEX=06) is caused, the instruction is terminated, and no further action is taken.

The 6-digit A operand is then read from memory. If its value is not equal to the Event Count field of the Event Structure (B), then the Comparison Flags are set to EQUAL, the instruction is terminated, and no further action is taken.

If the 6-digit operand is equal to the Event Count field of the Event Structure (B), then the processor-dependent state of the Event is examined. If it is in the Happened state, then the Comparison Flags are set to EQUAL, the instruction is terminated, and no further action is taken. If it is in the Not Happened state, then the following steps are taken.

1. The Event Waiter Link field of the Event Structure A is copied into the Next Task in List field of the Reinstate List entry for the current task.

2. The Current Task Number located in the currentReinstate List entry is stored into the Event Waiter Link field of the Event Structure (A).

3. The Waiting Event flag (02) is stored into the State Indicator field of the current task's Reinstate List entry.

4. A Failed Event flag (02) is stored into the Instruction Interrupt Cause Descriptor.

5. The Comparison Flags are set to LOW and an Instruction Interrupt is sent to the Kernel. The next instruction address is stored in the Interrupt Frame.

INTERRUPT (INT) OP = 90

Function

The Interrupt instruction is used to initiate the transfer of the system environment to the MCP Kernel. Information about the interrupt and its accompanying Kernel Request is stored in the Kernel Data Area, and then the Interrupt Procedure is invoked. The Interrupt Procedure is used by the processor hardware to accomplish the actual transfer of the system environment to the Kernel. (See Section 7, Interrupt Procedure.) The Interrupt instruction may only be executed in Privileged Mode.

Format

90	AF	BF	A

AF Field

The AF field is the length of the A data field. AF may specify an indirect field length, but a literal flag will cause an Invalid Instruction fault (IEX=21). A value of "00" indicates that no units are to be moved.

BF Field

The BF field contains the 8-bit Kernel Request code. This value may also be specified as indirect. (See Section 7, Kernel Requests.)

A Field

The A operand contains the address of the data field. The address may be indexed, indirect, or extended. The final address must be UN or an Invalid Instruction fault will occur (IEX=03).

Implementation

The Interrupt Descriptor value denoting an instruction-related interrupt, "06", is stored in the Instruction Interrupt Cause Descriptor field of the Kernel Data Area (absolute memory locations 32-33). (See Table 7-4, Kernel Data Area.)

The value of BF, the Kernel Request code, is stored in the MCP Kernel Request Code field of the Kernel Data Area (absolute memory locations 34-35).

If the value of AF is not equal to zero, then the A data field is stored in the MCP Kernel Request Data field of the Kernel Data Area (absolute memory addresses 8000-8039). If the value of AF exceeds "40", then an Invalid Instruction fault will be caused (IEX=25).

The Interrupt Procedure is then invoked, which will store the address of the next instruction to be executed in the Interrupt Frame.

I/O COMPLETE (IOC) OP = 98

Function

This privileged instruction tells how much data was moved on the I/O. It also decrements the I/Os in Process Count and resets the flag.

Format

98	AF	BF	A	B

AF Field

A length of 6 digits (6) must be specified directly or as an indirect field length or a literal.

BF Field

The BF field contains the Channel number. BF may have an indirect field length. BF can specify any octal value from "00" to "77", or the non-octal value "08". Use of any other Channel numbers causes an Invalid Instruction fault (IEX=26).

A Field

The A field contains the address of the 6-digit Memory Area Status Table (MAST). This address may be indexed, indirect, or extended. The final address controller must specify UN or an Invalid Instruction fault (IEX=03) is caused.

B Field

The B field contains the address of the 6-digit resultant field. This address may be indirect, indexed, or extended. The final address controller must specify UN or an Invalid Instruction fault (IEX=03) is caused.

Implementation

The I/O Complete instruction stores the unsigned difference between the I/O buffer begin and end address registers for the specified channel (BF) in memory (B). (The buffer begin address will have been incremented during the I/O operation to show the number of bytes transferred.) IOC examines the specified MAST entry (A), decrements the I/Os in Process field, and then sets the Comparison Flags accordingly. These steps are discussed in detail below.

- If the I/O Channel is busy, then the Comparison Flags are set to HIGH and the instruction is terminated with no further action.

If the I/O Channel is not busy, then the following steps are taken.

- If the Channel Number is an octal value in the range from "00" - "77", the difference between the end address and the begin address is stored at the specified memory location (B).

If the Channel Number is the non-octal value "08", then a value of zero is stored at the specified memory location (B).

- The MAST Number (A) is used as an array subscript into the MAST in order to locate the specified entry. If the MAST Number (A) is invalid, an Address Error fault (AEX=34) is caused and the instruction is terminated with no further action.

- The Number of I/Os in Process field, located in the MAST entry, is examined. If it is equal to zero, then an Invalid Instruction fault (IEX=05) is caused and the instruction is terminated with no further action. Otherwise, the value of this field is decremented by one.

- If the Number of I/Os in Process field is now equal to zero, the Status digit of the MAST entry is examined. If the Inhibited I/O Memory Area flag is set, then the Comparison flags are set to LOW and the instruction is terminated.

  If the Inhibited I/O Memory Area Flag is reset or if the value of the Number of I/Os in Process field is not equal to zero, then the Comparison Flags are set to EQUAL and the instruction is terminated.

LOAD INDEX REGISTERS (LIX) OP = 67

Function

This instruction allows the Mobile Index Registers (IX4, IX5, IX6, and IX7) and the IX1, IX2, and IX3 registers to be loaded.

Format

67 AF BF A

AF Field

The AF field is unused. AF may be indirect, but a literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

The BF field contains the Load variant, and may be specified as an indirect field length. The least significant digit of the BF field specifies the Base Indicant to be loaded as follows:

VARIANT	BFL
Base Indicant #7	7
Base Indicant #6	6
Base Indicant #5	5
Base Indicant #4	4
Base Indicant #3	3
Base Indicant #2	2
Base Indicant #1	1
No Base Indicant Value	0

The use of all other BFL values will cause an Invalid Instruction fault (IEX=26).

The most significant digit of the BF field specifies the Index Register to be loaded as follows:

VARIANT	BFM
Load Index Register #7	7
Load Index Register #6	6
Load Index Register #5	5
Load Index Register #4	4
Load Index Register #3	3
Load Index Register #2	2
Load Index Register #1	1
Load Mobil Index Registers (4)	0

The use of all other BFM values is reserved and will cause an Invalid Instruction fault (IEX=26).

A Field

The A field contains the address of the Index Register field. This address may be indexed, direct or extended. The final address controller must equal UN or and Invalid Instruction fault (IEX=03) will be caused.

Implementation

This instruction provides the memory address A of the starting location of either:

- an 8-digit field that contains the value that is to be loaded into the specified Index Register (BFM), or
- the 32-digit field that contains the values that are to be loaded into the four Mobile Index Registers (IX4, IX5, IX6, and IX7).

Each 8-digit byte represents an index register according to the following format.

INFORMATION	DIGITS
Sign	00
Base Indicant	01
Address	02-07

If the load variant (BFM) specifies the load of a single register and if a Base Indicant is specified, the value contained in the LSD of BF is inserted into the Base Indicant digit of the specified Index Register.

The Mobile Index Registers are registers in the hardware rather than in main memory. When the remaining registers (IX1, IX2, and IX3) are loaded, the associated memory location (8, 16, and 24 relative to Base #0) will be updated to the value found in the A operand.

LOCK/UNLOCK (LOK) OP = 60

Function

The privileged LOK instruction allows the operating system to preserve the integrity of a shared data structure. LOK locks other users out while a data structure is being accessed by a task.

Format

60 AF BF A

AF Field

The AF field is unused and reserved. AF may be indirect, but a literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

BF is an Instruction variant and may be indirect. All values other than those listed in the following table are reserved and cause an Invalid Instruction fault (IEX=26).

VARIANT	FUNCTION
09	Event Cause and Reset
08	Event Reset and Wait
07	Test Happened Status
06	Event Reset
05	Event Wait
04	Event Cause
02	Conditional Lock
01	Unconditional Lock
00	Unlock

A Field

The A field contains the address of the Lock/Event Structure. This address can be indexed, indirect, or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) is caused.

If BF has a value in the range of 00 - 02, then A represents a Lock Structure.

If BF has a value in the range of 04 - 09, then A represents an Event Structure.

The Lock Structure format is shown below.

INFORMATION	DIGITS
Lock Status field	00 - 01
Lock Owner field	02 - 05
Lock Waiter Link field	06 - 09
Lock Number field	10 - 13
Lock Number Link field	14 - 17
Reserved	18 - 19

The Event Structure format is shown below.

INFORMATION	DIGITS
Event Status field	00 - 01
Event State field	02 - 05
Event Waiter Link field	06 - 09
Event Designator field	10 - 13
Event Count field	14 - 19

If any of the Lock/Event Structure values are invalid, an Invalid Instruction fault (IEX=07) is caused.

IMPLEMENTATION

LOK examines the Lock/Event Structure A and, according to its value and the instruction variant BF, modifies, if necessary, the Structure A and the associated lock fields within the Reinstate List entry for the current task and other tasks owning or contending for the lock or event.

The processor must determine if a lock is owned or available. If the Owner field of the Lock Structure is equal to zero, the lock is available. Otherwise, the lock is owned.

The processor must determine if the Event has happened. If the Event State field contains all zeroes, the event has happened. If it contains all hexadecimal "F"s, then the event has not happened. If the Event State field contains any other value, then an Invalid Instruction fault (IEX=06) is caused, the instruction is terminated, and no further action is taken.

The Case Where BF = 00 (UNLOCK)

This variant releases a lock and, if any task is waiting for this lock, causes an interrupt to the MCP Kernel.

The Lock Structure A is read from memory. The value of the Lock Owner field must equal the Current Task Number of the Kernel Data Area, or an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

Zeroes are stored into the Lock Owner field of A to indicate that this lock is now available.

The Lock Number field of A is compared to the MCP Canonical Lock Number field located in the Reinstate List entry for the current task. If the two values are not equal, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

If the number fields are equal, then the contents of the Lock Number Link field of Lock Structure A is stored into the MCP Canonical Lock Number field of the Reinstate List entry for the current task.

If the Lock Waiter field of Lock Structure A is equal to zero, then the Comparison Flags are set to EQUAL and the instruction is terminated.

If the Lock Waiter field of Lock Structure A is not equal to zero, then the following steps are taken.

1. The Reinstate List pointer has been located with a Write Hardware Register instruction (OP=65, BF=00). The 4-digit Lock Waiter field of A is used as an array subscript into this table to locate a new Reinstate List entry. A sign and Base Indicant character of "C7" and 6 digits of the absolute address of this Reinstate List entry are stored in address 24 - 31 relative to the Kernel Data Area (IX3).

2. The Release Lock flag (03) is stored into the Instruction Interrupt Cause Descriptor field of the Kernel Data Area.

3. The Comparison Flags are set to HIGH and an Interrupt is issued with the address of the next instruction to be executed stored in the Interrupt Frame.

The Case Where BF = 01 (UNCONDITIONAL LOCK)

This variant competes for the lock specified by the Lock Structure A and, if the lock is owned, causes an interrupt to the MCP Kernel.

The Lock Structure A is read from memory.

The Lock Number field of A is compared with the MCP Canonical Lock Number field, located in the Reinstate List entry for the current task. If the Lock Number field value is less than or equal to the MCP Canonical Lock Number field value, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

If the lock is available, then the 4-digit Current Task Number (in the Kernek Data Area) is stored into the Lock Owner field of A and the following procedure is executed.

1. The contents of the MCP Canonical Lock Number field are copied into the Lock Number Link field of A.

2. The contents of A's Lock Number field are copied into the MCP Canonical Lock Number field of the Reinstate List entry for the current task.

3. The Comparison Flags are set to EQUAL and the instruction is terminated.

If the lock is owned, the following procedure is executed.

1. A's Lock Owner field is copied into the Task Number Owning field of the Reinstate List entry for the current task.
2. A's Lock Waiter Link field is copied into the Next Task in List field of the Reinstate List entry for the current task.
3. A's 4-digit Lock Owner field is used as an array subsript into the Reinstate List in order to locate a new entry. A sign and Base Indicant character of "C7" and six digits of the absolute address of this Reinstate List entry are stored in address 24 - 31 of the Kernel Data Area (IX3).
4. The Kernel Data Area's Current Task Number field value is stored into A's Lock Waiter Link field.
5. The Waiting Lock flag (01) is stored into the State Indicator field of the current task's Reinstate List entry.
6. The Failed Lock flag (01) is stored into the Instruction Interrupt Cause Descriptor field of the Kernel Data Area.
7. The Trace Mode Bit is ignored.
8. The Comparison Flags are set to LOW and an interrupt is issued to the Kernel that stores the current instruction address in the Interrupt Frame.

The Case Where BF = 02 (CONDITIONAL LOCK)

This variant attempts to obtain the lock specified by the Lock Structure A.

If the lock is available, the following steps are performed.

1. If the Lock Number field value of A is less than or equal to the the MCP Canonical Lock Number field value for the current task's Reinstate List entry, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.
2. If the Lock Number field value is greater, then the current task's 4-digit Task Number is stored into the Lock Owner field of A and zeroes are stored into the Lock Waiter Link field of A.
3. The contents of the MCP Canonical Lock Number field are stored into the Lock Number Link field of A.
4. The contents of A's Lock Number field are copied into the MCP Canonical Lock Number field.
5. The Comparison Flags are set to EQUAL and the instruction is terminated.

If the lock is owned, the following steps are taken.

1. If the Lock Owner field equals the Task Number, then the Comparison Flags are set to LOW and the instruction is terminated with no further action.
2. If the owner is not the current task, then the Comparison Flags are set to HIGH and the instruction is terminated with no further action.

The Case Where BF = 04 (EVENT CAUSE)

This variant causes an event and signals this accomplishment to all tasks that are waiting for this event.

The Event Structure (A) is read from memory. If the Event Designator field is not equal to zero, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

If the event is in the Happened state, the Event Count field is incremented by one. (If this field value was originally set to the maximum value for the container, then it is set to zero.) The Comparison Flags are set to EQUAL and the instruction is terminated with no further action.

If the event is in the Not Happened state, then the Event Count field is incremented by one. (If this field value was originally set to the maximum value, then it is set to zero.) The Event State field is then set to the Happened state.

The Event Waiter Link field is then examined. If it is equal to zero, then the Comparison Flags are set to EQUAL and the instruction is terminated with no further action.

If the Event Waiter Link field is not zero, then the following steps are taken.

1. The 4-digit Event Waiter Link field of A is used as an array subscript into the Reinstate List in order to locate the new entry. A sign and base indicant character of "C7" and the 6-digit absolute address of this Reinstate List entry are stored at address 24 - 31 relative to the Kernel Data Area.

2. The Released Event flag (04) is stored into the Instruction Interrupt Descriptor at address 32 - 33 of the Kernel Data Area.

3. The Comparison Flags are set to HIGH and an Interrupt Instruction (OP=90) is issued to the Kernel. This instruction stores the address of the next instruction to be executed in the Interrupt Frame.

The Case Where BF = 05 (EVENT WAIT)

This variant causes the current task to wait until the specified event is caused, if it is currently in the Not Happened state.

The Event Structure A is read from memory. If the Event Designator field is not equal to zero, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

If the event is in the Happened state, the Comparison Flags are set to EQUAL and the instruction is terminated with no further action.

If the event is in the Not Happened state, then the following steps are taken.

1. The Event Waiter Link field is copied into the Next Task in List field located in the Reinstate List entry for the current task.

2. The current Task Number is stored into A's Event Waiter Link field.

3. The Waiting Event flag (02) is stored into the State Indicator field located in the Reinstate List entry for the current task.

4. A Failed Event flag (02) is stored into the Instruction Interrupt Cause Descriptor field at address 32 - 33 of the Kernel Data Area.

5. The Comparison Flags are set to LOW and an Instruction Interrupt (OP=90) is made to the Kernel. The next instruction address is stored in the Interrupt Frame.

The Case Where BF = 06 (EVENT RESET)

This variant resets the Happened state of the event to the Not Happened state.

The Event Structure A is read from memory. If the Event Designator field is not equal to zero, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

If the event is in the Not Happened state, then the Comparison Flags are set to HIGH and the instruction is terminated with no further action.

If the event is in the Happened state, then the Event State field is reset to the Not Happened state and zeroes are stored into A's Event Waiter Link field.

The Comparison Flags are set to EQUAL and the instruction is terminated.

The Case Where BF = 07 (EVENT TEST HAPPENED STATUS)

This variant tests whether or not an event happened.

The Event Structure A is read from memory. If the Event Designator field is not equal to zero, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

If the event is in the Not Happened state, the Comparison Flags are set to HIGH and the instruction is terminated with no further action.

If the event is in the Happened state, then the Comparison Flags are set to EQUAL and the instruction is terminated with no further action.

The Case Where BF = 08 (EVENT RESET AND WAIT)

This variant resets an Event Structure to the Not Happened state and forces the current task to be suspended (wait) until the event has been caused.

The Event Structure A is read from memory. If the Event Designator field is not equal to zero, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

The Event State field is reset to the Not Happened state, and the following steps are taken.

1. A's Event Waiter Link field is copied into the Next Task in List field of the Reinstate List entry for the current task.

2. The Current Task Number located in the current task's Reinstate List entry is copied into A's Event Waiter Link field.

3. The Waiting Event flag (02) is stored into the State Indicator field of the current task's Reinstate List entry.

4. The Failed Event flag (02) is stored into the Instruction Interrupt Cause Descriptor field at address 32 - 33 of the Kernel Data Area.

5. The Comparison Flags are set to LOW and an Instruction Interrupt is made to the Kernel. The next instruction address is stored in the Interrupt Frame.

The Case Where BF = 09 (EVENT CAUSE AND RESET)

This variant causes an event and thus allows any tasks that have been waiting on the event to continue processing. The Event Structure is left in the Not Happened state.

The Event Structure A is read from memory. If the Event Designator field is not equal to zero, then an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated with no further action.

The Event Count field is incremented by one and the Event State field is reset to the Not Happened state.

If A's Event Waiter Link field is equal to zero, then the Comparison Flags are set to EQUAL and the instruction is terminated with no further action.

If the Event Waiter Link field is not equal to zero, then the following steps are taken.

1. The 4-digit Event Waiter Link field of A is used as an array subscript into the Reinstate List in order to locate the new entry. A sign and base indicant character of "C7" and the 6-digit absolute address of this Reinstate List entry are stored at address 24 - 31 relative to the Kernel Data Area (IX3).

2. The Released Event flag (04) is stored into the Instruction Interrupt Descriptor at address 32 - 33 of the Kernel Data Area.

3. The Comparison Flags are set to HIGH and an Interrupt Instruction (OP=90) is issued to the Kernel. This instruction stores the address of the next instruction to be executed in the Interrupt Frame.

MOVE LOCK STRUCTURES (MLS) OP = 6A

Function

This privileged instruction moves a Lock Owner from a Lock Structure, or an Event Count from an Event Structure, to a destination field.

Format

6A	AF	BF	A	B

AF Field

The AF field is unused and reserved, but can be specified as an indirect field length. A literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

The BF field contains an Instruction variant and can be indirect.

VARIANT	FUNCTION
01	Move Lock Owner
00	Move Event Count

A Field

The A field contains the address of the Lock Structure or the Event Structure. This address can be indexed, indirect, or extended. The final address controller must be UN or an Invalid Instruction fault (IEX=03) will be caused.

B Field

The B field contains the address of the receiving field for the information being moved from the Lock Structure or the Event Structure. This address can be indexed, indirect, or extended. The final address controller must be UN or an Invalid Instruction fault (IEX=03) will be caused.

IMPLEMENTATION

The Case Where BF=00 (MOVE EVENT COUNT)

The Event Structure (A) is read from memory. If the Event Designator field is not equal to zero, or if the Event Count contains undigits, then an Invalid Instruction fault (IEX=06 or IEX=07, respectively) is caused and the instruction is terminated with no further action.

Event A's 6-digit Event Count field is moved to the 6-digit destination field specified by the B address.

The Case Where BF=01 (MOVE LOCK OWNER)

The Lock Structure (A) is read from memory. If the Lock Number field is equal to zero, or if it contains undigits, then an Invalid Instruction fault (IEX= 06 or IEX=07, respectively) is caused and the instruction is terminated with no further action.

Otherwise, Lock A's 4-digit Lock Owner field is moved to the 4-digit destination field specified by the B address.

MEASUREMENT OP (MOP) OP = 87

Function

The MOP instruction is used to monitor addresses, procedures, or other elements of interest. MOP is followed by a bit pattern that is sent to one the backplanes when it is encountered by the processor. A hardware monitor connected to the backplane pins can then be used to record the number of times different MOPs are executed.

Format

87 AF BF A B

AF Field

A length of six (6) must be specified directly or as an indirect field length or literal.

BF Field

A length of six (6) must be specified directly or as an indirect field length.

A Field

This field is the address of the Setting field. This address can be indexed indirect or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) is caused.

B Field

This field is the address of the Mask field. This address can be indexed, indirect or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) is caused.

The Measurement Register

The Measurement Register is an 8-digit register with outputs that are made available as external outputs of the processor so that they can be monitored by various hardware monitoring devices.

The format of the Measurement Register is shown below.

INFORMATION	DIGITS
User	00 - 05
Name	06 - 07

The MOPOK signal, available externally, is held to "zero" at any time that the Measurement Resister is being changed. It is held to a "one" at all other times.

IMPLEMENTATION

The Measurement instruction is used to load the User portion of the Measurement Register. The 6-digit Mask field is read from memory location B and used to determine which bits in the Measurement Register are capable of being changed by the Setting field.

Each Mask field bit that is set to "one" permits the corresponding bit in the Measurement Register's User field to assume the state of the corresponding bit in the Setting field (A).

Each Mask field bit that is set to "zero" prevents changes to the corresponding bit in the Measurement Register's User field.

The Measurement Register is changed by the Virtual Branch Reinstate (OP=93), Branch Communicate (OP=30), Hyper Call (OP=62), and Return (OP=63) instructions, as well as by the Interrupt and Hardware Call procedures.

MOVE STRING (MVS) OP = A0

Function

This instruction moves a specified substring from a specified source to a specified destination.

FORMAT

A0 AF BF A B

AF Field

The AF field is the Source Field variant. AF can be indirect, but a literal flag causes an Invalid Instruction fault (IEX=21).

The least significant digit is the Update variant.

- A value of "0" indicates that no update should take place.
- A value of "1" indicates that an update of the source string begin address should take place.
- The use of all other LSD AF values is reserved and causes an Invalid Instruction fault (IEX=25).

The most significant digit is the Memory Area variant.

- A value of "0" indicates that the Memory Area specified by the Environment Number and the Memory Area Number contained in the Source String Descriptor A is to be used for the addresses contained within the Source String Descriptor, A.
- A value of "4" indicates that the Memory Area specified for the Source String Descriptor A is to be used for the addresses contained within the Source String Descriptor.
- The use of all other MSD AF values is reserved and causes an Invalid Instruction fault (IEX=25).

BF Field

The BF field is the Destination Field variant. BF can be indirect.

The two least significant bits of the most significant digit of BF contain the Substring Select variant and are coded to select a substring from the destination string. The possible selections are shown below.

SUB STRING RANGE		VALUE
Container Begin —>	Container End	3
Reserved		2
String End —>	Container End	1
String Begin —>	String End	0

The "4" bit of BF's most significant digit is the Memory Area variant.

- A value of "0" indicates that the Memory Area specified by the Environment Number and the Memory Area Number contained in the Destination String Descriptor B is to be used for the addresses contained within the Destination String Descriptor.
- A value of "4" indicates that the Memory Area specified for the Destination String Descriptor B is to be used for the addresses contained within the Destination String Descriptor.

BF's least significant digit is the Update variant.

- A value of "0" indicates that no update should take place.
- A value of "1" indicates that an update should take place.

The use of all other BF values is reserved and causes an Invalid Instruction fault (IEX=26).

A Field

The A field contains the address of the Source String Descriptor. This address can be indexed, indirect, or extended. The final address controller must specify UN or an Invalid Instruction fault (IEX=03) is caused.

B Field

The B field contains the address of the Destination String Descriptor. This address can be indexed, indirect, or extended. The final address controller specifies the padding variant as shown below.

PADDING VARIANT	B-CONTROLLER
Pad with zero	0 (UN)
No padding	1 (SN)
Pad with blank (40)	2 (UA)

IMPLEMENTATION

This instruction moves the string begin to string end substring from the location specified by the Source String Descriptor A to the location specified by the Destination String Descriptor B and the String Select variant BF.

Any padding that may be required in the destination has its type defined in the Padding variant. Data is not read from a string end address or written into a string end address or the container end address.

If the source substring begin address is greater than the source substring end address, then an Address Error fault (AEX=01) is caused and the instruction is terminated without further action.

If the destination substring begin address is greater than the destination substring end address, then an Address Error fault (AEX=01) is caused and the instruction is terminated without further action.

If AF's Update bit is equal to "0", then the Source String Descriptor is not changed. If it is equal to "1", then the Source String Descriptor's string begin address is set to point to one digit beyond the last digit moved.

If BF's Update bit is equal to "1", then the Destination String Descriptor's string end address is set to point to one digit beyond the last digit written. Furthermore, if the Substring Select bits are equal to "3" (Container begin to Container end addresses), then the Destination String Descriptor's string begin address is set to the same value as the Container begin address.

Equal Source and Destination Lengths

In this case, the source substring is moved to the destination substring and the Comparison Flags are set to EQUAL.

Source Length is Longer Than the Destination Length

A length corresponding to the destination length is moved from the left-justified source substring to the destination substring. The Comparison Flags are set to HIGH.

Source Length is Shorter Than the Destination Length

A length corresponding to the source length is moved from the source substring to the left-justified destination substring. The B address controller bits determine if and what kind of padding is required.

If the B address controller is equal to a "1", then no padding takes place and the Comparison Flags are set to LOW.

If the B address controller is equal to a "0", then the remaining digits in the destination string are padded with zeroes and the Comparison Flags are set to EQUAL.

If the B address controller is equal to a "2", then the remaining digits in the destination string are padded with blanks and the Comparison Flags are set to EQUAL. If there are an odd number of digits remaining in the destination string, then an Invalid Instruction fault (IEX=07) is caused and the instruction is terminated without updating the the pointers.

Null Strings

If the source string is null, then the destination field is filled with the padding character specified by the B address controller.

If the destination string is null, then the Comparison Flags are set to HIGH and the instruction is terminated with no further action.

If both strings are null strings, then the Comparison Flags are set to EQUAL and the instruction is terminated with no further action.

Overlap

String containers and descriptors that occupy any of the same memory locations, partially overlapping string containers, and partially overlapping string descriptors produce unspecified results that may vary from processor model to processor model.

Partially overlapping substrings produce unspecified results unless the destination substring select bits equal "3" and the descriptors totally overlap. In this case, the string is normalized by moving the string to the left and filling the container with the padding characters specified by the B address controller.

Total overlap of string containers and descriptors is allowed.

RETURN (RET) OP = 63

Function

RET is a companion instruction to the Hyper Call (HCL) and Virtual Enter (VEN) instructions and the Hardware Call Procedure. An unconditional branch is made to the location specified in the Stack Frame as the next program instruction.

Format

63 AFBF

AFBF Field

The AFBF field is unused and reserved.

IMPLEMENTATION

RET reverses the action of the calling instruction or procedure by:

- loading the machine state from the current stack,
- restoring the user environment, and
- executing an unconditional branch to the location specified in the Stack Frame's Next Program Instruction.

The value of IX3+14 represents the address (relative to Base #0) of the 2-digit Stack Frame Indicator. The type of calling procedure and the type of return procedure to be executed are stored here, as follows.

INFORMATION	INDICATOR
Virtual Enter/Virtual Exit	FF
Hyper Call/Hyper Return	FE
Hardware Call/Return	FD

All other Stack Frame Indicator values are invalid, and will cause an Invalid Instruction fault (IEX=37).

The parameter values stored in the stack are unchanged by this instruction and are not copied into any other area of memory.

Virtual Enter/Virtual Exit

IX3 minus 14 represents the address of the Virtual Enter Stack Frame relative to Base #0. The information in the Virtual Enter Stack Frame is used to replace the respective state in the machine. It is stored in the following sequence.

INFORMATION		DIGITS
Old TOS —>	Measurement Register (User)	00 - 05
	COM and OVF Flags	06 - 07
	Active Environment Number	08 - 13
New IX3 —>	Next Instruction Address	14 - 19
	Saved IX3 Value	20 - 27
	Stack Frame Indicator ("FF")	28 - 29
	Stack Parameters (0 - 9999 bytes)	
New TOS —>		

The address in location 40, relative to Base #0, is replaced with the value of IX3 minus 14, relative to Base #0. After the state is loaded, the contents of IX3 are replaced with Virtual Enter Stack Frame's IX3 value.

The Mode Indicators, Accumulator, and Interrupt Mask Register are not changed by this variant.

If the Environment Number contained in the Stack is equal to zero, then the environment being returned to is the same environment that is specified by the Active Environment Number. Since the correct Base/Limit pairs are already resident within the processor, then the following step is unnecessary.

If the Environment Number in the Stack is not equal to zero, then this instruction is a Non-Local Virtual Exit. In this case, the new Memory Area Table (pointed to by this Environment Number) must be loaded.

The next instruction address, relative to Base #1, is then resolved, and an unconditional branch is executed to that address.

Hyper Call/Hyper Return And Hardware Call/Return

This variant can only be executed in Privileged Mode.

IX3 minus 80 represents the address of the Hyper Call or Hardware Call Stack Frame, relative to Base #0. The information in the Hyper Call/Hardware Call Stack Frame is used to replace the respective state in the machine.

The Hyper Call/Hardware Call Stack Frame is stored in the following sequence.

INFORMATION		DIGITS
New TOS —>	Accumulator	00 - 27
	Measurement Register (User)	28 - 35
	Interrupt Mask	36 - 37
	Mobile Index Registers	38 - 69
	Mode Indicators	70 - 71
	COM and OVF Flags	72 - 73
	Active Environment Number	74 - 79
Old IX3 —>	Next Instruction Address	80 - 85
	Saved IX3 Value	86 - 93
	Stack Frame Indicator (FE/FD)	94 - 95
	Stack Parameters (0 - 9999 bytes)	
Old TOS —>		

The address in location 40, relative to Base #0, is replaced with the value of IX3 minus 80, relative to Base #0. After the state is loaded, the contents of IX3 are replaced with the value of IX3 in the Hyper Call/Hardware Call Stack Frame.

The MOPOK line is set to "zero" while the Measurement Register is being changed and is set to "one" at all other times.

The Memory Area Table pointed to by the Environment Number in the Stack Frame is loaded.

The Soft Fault and Trace Mode Fault Condition Indicators are examined in order to determine if a Hardware Call Procedure should be executed. If Soft Fault is enabled and the Soft Fault Pending Flag of the task's Reinstate List entry is not equal to zero, then a Soft Fault Condition exists.

If this variant is a Hyper Return (indicated by a Stack Frame Indicator value of FE) and Trace Mode is enabled, then a Trace Fault Condition exists.

If this variant is a Hardware Return (indicated by a Stack Frame Indicator value of FD), then the Trace Mode is ignored until the execution of the following instruction has been completed.

If a Fault Condition has been found, a Hardware Call Procedure is executed that will store the address of the next instruction to be executed and report all existing Fault Conditions.

Otherwise, the next instruction relative to Base #1 is resolved with the new Base/Limit information. Finally, an unconditional branch to that address is executed.

STORE INDEX REGISTERS (SIX) OP = 68

Function

This instruction allows the four Mobile Index Registers (IX4, IX5, IX6, and IX7), as well as the IX1, IX2, and IX3 registers to be stored.

Format

68 AF BF A

AF Field

The AF field is unused and reserved. AF can be indirect, but a literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

The BF field contains the Store variant, and can be specified as an indirect field length. The following variants can be specified by this after any indirect field length has been resolved.

VARIANT	BFM
Store Index Register #7	7
Store Index Register #6	6
Store Index Register #5	5
Store Index Register #4	4
Store Index Register #3	3
Store Index Register #2	2
Store Index Register #1	1
Store Mobile Index Registers	0

The use of all other BF values will cause an Invalid Instruction fault (IEX=26).

A Field

The A field contains the address of the Index Register field. This address can be indexed, direct or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) will be caused.

IMPLEMENTATION

This instruction provides the memory address A of the starting location of either:

- an 8-digit field that will be used to store the specified BFM Index Register, or
- a 32-digit field that will be used to store the four Mobil Index Registers (IX4, IX5, IX6, and IX7)

Each 8-digit byte represents an index register according to the following format.

INFORMATION	DIGITS
Sign	00
Base Indicant	01
Address	02-07

SEARCH LIST (SLT) OP = 64

Function

SLT is a general search instruction for linked lists.

Format

64 AF BF A B C

AF Field

The AF field is reserved and unused. AF can be specified as an indirect field length, but a literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

BF is a Search variant, and can be specified as an indirect field length. The following variants can be specified by this field after any Indirect Field Length has been resolved.

FUNCTION	BF MOST SIGNIFICANT DIGIT
Store IX2 (Delink)	4
Normal (IX1 only)	0

FUNCTION	BF LEAST SIGNIFICANT DIGIT
Search Lowest	9
Search Highest	8
No Bit Equal	7
Any Bit Equal	6
A .GE. B	5
A .GT. B	4
A .LE. B	3
A .LT. B	2
A .NE. B	1
A .EQ. B	0

The use of all other BF values is reserved, and will cause an Invalid Instruction fault (IEX=26). The individual comparison details are discussed later on in this section.

A Field

The A field contains the address of the key field. The address can be indexed, indirect, or extended. The final address controller specifies the data type for both the key, A, and the comparison. The address controller must specify UN or UA. An SN controller will cause an Invalid Instruction fault (IEX=03).

B Field

The B field contains the address of the list field entry pointer. This address can be indexed, indirect, or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) will be caused. This 6-digit field contains an address that is relative to the same memory area as the B address. This address is a pointer to the first list to be compared. A value of "EEEEEE" indicates an empty (null) list.

C Field

The C field contains the address of the list descriptor. This address can be indexed, indirect, or extended. The final address controller must equal UN or and Invalid Instruction fault (IEX=03) will be caused. The format of this 18-digit field is shown below.

INFORMATION	DIGITS
Link Offset	00 - 05
Comparison Offset	06 - 11
Key Length	2 - 17

All of the list descriptor values are digits. If any of the values are undigits, an Address Error fault (AEX=34) will be caused.

IMPLEMENTATION

The list to be compared can be empty or not empty. The steps to be taken in both cases are covered below.

An Empty List

An empty list is indicated by a value of "EEEEEE" in the list field entry pointer B. The value "CiEEEEEE" is stored in the IX1 register, where "i" represents the Base Indicant of the resolved B operand. The Comparison Flags are set to NULL.

If the store of IX2 is specified by the MSD of BF, then the address of the list field entry pointer B (relative to the same base as the resolved B operand) is stored in IX2.

A Non-Empty List

List data, pointed to by the key A, has its length recorded in the key length field of list descriptor address C. This data is compared with the data located in a specifies a location in memory that contains the 6-digit address of the list field entry pointer in memory. This address is relative to the same memory area as the resolved B operand.

The field key is found by adding the comparison offset (digits 06-11 within field C) to the value of the list field entry pointer.

The result of the comparison will be one of two results: either the comparison condition, as specified in the LSD of BF, will be met, or if will not. Subsequent actions for each case are discussed below.

The Comparison Condition is Met

Except in the case of Search Highest or Search Lowest, the list field entry pointer, relative to the same base as the B operand, is stored in IX1.

In the case of Search Highest and Search Lowest, the entry list is examined below the address of the highest or lowest value (as is appropriate) is stored in the IX1 register.

If the store of IX2 is specified by the MSD of BF and this is the first comparison, the address of the list field entry pointer is stored in IX2. (This address is relative to the same base as is the resolved B operand).

If it is other than the first comparison, the address of the previous link address field (that is, the list field entry pointer plus the link offset) is stored in IX2. (Again, this address is relative to the same base as is the resolved B operand).

The Comparison Condition is Not Met

The sum of the list field entry pointer and the link offset (C 00:6) is used as an address to obtain the 6-digit link address of the next field entry pointer from memory. This procedure is repeated until either the comparison condition is met, or until a link address has the value "EEEEEE". That is, the list is traversed and tested until either the comparison is met, or the list's end is reached.

If indeed the value "EEEEEE" is encountered, the null list value "CiEEEEEE" is stored in IX1. If the store of IX2 is specified by the MSD of BF, then the address of the link address field belonging to the list's last entry is stored in IX2. (This address is equal to the sum of the list field entry point plus the link offset, and is expressed relative to the same base as is the resolved B address).

The tests preformed in order to make different comparisons are discussed below.

Any Bit Equal

The key field A must be ANDed with the comparison field to determine if the result is equal to zero. If the result is not equal to zero, a match occurred.

Search Lowest

The Search Lowest search is terminated only when a null link is reached. The entire list is searched for the lowest comparison field that is lower than key A. If at least one entry is found, then this entry's address is stored in IX1 with the same base indicant as that of the resolved B operand.

If the store of IX2 is specified by the MSD of BF, then the address of the previous link address field is stored in IX2.

If no entries are found that are less then the key A, then the null list value "CiEEEEEE" is stored in IX1. If the store of IX2 is specified by the MSD of BF, then the address of the last list entry's link address field is stored in IX2. (This address is equal to the sum of list field entry point plus the link offset). As always, the address stored in IX2 for the above two cases is expressed relative to the same base as that of the resolved B operand.

Search Highest

The Search Highest search is only terminated when a null link is encountered. In a similar fashion to the Search Lowest search, the entire list is searched for the highest comparison field entry that is also higher then the key A. If one such entry is found, then its address is stored in IX1, with the same base indicant as that of the resolved B address.

If the store of IX2 is specified by the MSD of BF, then the address of the previous link address field is stored in IX2. (This address is equal to the sum of the list field entry point plus the link offset, and is expressed relative to the same base as that of the resolved B address).

If no entries are found that are greater than the key A, then the null list value "CiEEEEEE" is stored in IX1. If the store of IX2 is specified by the MSD of BF, then the address of the link entry is stored in IX2. (This address is equal to the sum of the list field entry point plus the link offset, and is expressed relative to the same base as is the resolved B address.)

The list must reside within one memory area as specified by the Base Indicant associated with the B address. The processor will not check for improper memory assignments.

No Bit Equal

All of the key field A must be ANDed with all of comparison field. If the result is equal to zero, a match was found.

Comparison Flags

If the comparison condition is met on the first entry, then the Comparison Flags are set to LOW. If the comparison condition is met on other then first entry, the Comparison Flags are set to EQUAL. If the condition is not met, the Comparison Flags will be set to the HIGH. If the list is empty, then the Comparison Flags will be set to NULL.

SYSTEM STATUS (SST) OP = 99

Function

This privileged instruction gives the status of the machine including memory area reports, Snap Picture, over-temperature, processor type, number, system number, serial number.

Format

99 AF B A

AF Field

The AF field is unused and reserved. It can be specified as an indirect field length. A literal flag causes an Invalid Instruction fault (IEX=21).

BF Field

BF is a Status variant that can be specified as an indirect field length. The use of all other BF values than those listed below is reserved and causes an Invalid Instruction fault (IEX=26).

VARIANT	FUNCTION
00	Status Indicators
01	System ID

A Field

This field contains the address of the Status data field. This address can be indexed, indirect, or extended. The final address controller must equal UA or an Invalid Instruction fault (IEX=03) is caused.

IMPLEMENTATION

This privileged instruction stores the number of bytes of status, as specified below, into the specified memory location (A).

The Case Where BF = 00 (SYSTEM STATUS)

The system status is stored in memory in the format below.

INFORMATION	BYTE	BIT
Reserved (0)	00	4-7
Memory Error Report Status	00	3
Reserved	00	2
Temperature Warning Status	00	1
Voltage Warning Status	00	0
Machine Dependent Data	01-99	ALL

The Memory Error Report Status bit indicates that a Memory Error Report has been stored in memory at a location that has previously been set with a Write Hardware Register instruction (OP=65, BF=02).

The Voltage Warning Status bit is set to indicate that the system input voltage is less than a preset value. This condition does not cause a power-off cycle. The bit remains "true" as long as the warning condition exists.

The Temperature Warning Status bit indicates that the system temperature has exceeded a preset value. This condition does not cause a power-off cycle. The bit remains "true" as long as the warning condition exists.

The Case Where BF = 01 (SYSTEM ID)

The System ID is stored in memory in the format below.

INFORMATION	EBCDIC BYTES
Processor Type	00 - 09
Specification Level	10 - 19
Shared System Number	20 - 21
Multiple Processor Number	22 - 23
Serial Number	24 - 33
Memory Size	34 - 49
Firmware Level	50 - 97
Reserved (00 00)	98 - 99

SEARCH TABLE (STB) OP = 66

Function

SLT is a general search instruction for tables.

Format

66 AF BF A B C

AF Field

The AF field is reserved and unused. AF can be specified as an indirect field length, but a literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

BF is a Search variant, and can be specified as an indirect field length. The following variants can be specified by this field after any Indirect Field Length has been resolved.

FUNCTION	BF
Search Lowest	09
Search Highest	08
No Bit Equal	07
Any Bit Equal	06
A .GE. B	05
A .GT. B	04
A .LE. B	03
A .LT. B	02
A .NE. B	01
A .EQ. B	00

The use of all other BF values is reserved, and will cause an Invalid Instruction fault (IEX=26). The individual comparison details are discussed later in this section.

A Field

The A field contains the address of the key field. The address can be indexed, indirect, or extended. The final address controller specifies the data type for both the key, A, and the comparison. The address controller must specify UN or UA. An SN controller will cause an Invalid Instruction fault (IEX=03).

B Field

The B field contains the address of the base table. This address can be indexed, indirect, or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) will be caused.

C Field

The C field contains the address of the table descriptor. This address can be indexed, indirect, or extended. The final address controller must equal UN or and Invalid Instruction fault (IEX=03) will be caused. The format of this 24-digit field is shown below.

INFORMATION	DIGITS
Table Entry Length (digits)	00 - 05
Comparison Offset (digits)	06 - 11
Key Length (digits)	12 - 17
Table Limit (address)	18 - 23

Invalid table descriptor values cause an Invalid Instruction fault (IEX=07).

IMPLEMENTATION

The data contained in the key A is compared to the data contained in a specified table field.

The value of the Comparison offset is added to the Table Base address B in order to identify the first field to be compared.

Except for Search Lowest or Search Highest, if the comparison condition is met, the address of the table entry is stored in IX1 and the Comparison Flags are set.

If the comparison condition is not met, the next table entry is examined. The sum of the Table Base Address (B) and the Table Entry Length (C 00:6) replaces the Table Base address to point at the next table entry. If the next table entry address is equal to or exceeds the Table Limit address (C 18:6) then the Comparison Flags are set to HIGH and the instruction is terminated. Otherwise, another comparison is executed using the data from the new table entry.

In the case of Search Lowest or Search Highest, the entire table is examined before the address of the entry with the lowest or highest value (as is appropriate) is stored in IX1.

The value of the Comparison offset is added to the Table Base address (B) in order to identify the first field to be compared.

If this address is greater than or equal to the Table Limit, then the table is empty. The value "CiEEEEEE" is stored in IX1, where "i" represents the Base Indicant of the resolved B address. The Comparison Flags are then set to NULL.

Except in the cases of Search Highest or Search Lowest, if the Table Limit is reached, the Comparison Flags are set to HIGH and the instruction is terminated as above with IX1 containing "CiEEEEEE".

The relative address stored in IX1 contains the Base Indicant associated with the resolved B address.

The table must reside within one memory area as specified by the Base Indicant associated with the resolved B address. The processor does not check for improper memory assignments.

Any Bit Equal

All of the A key field is logically ANDed with all of the comparison field to determine if the result is equal to zero. If the result is not equal to zero, a match occurred.

Search Lowest

The Search Lowest search is terminated only when the Table Limit is exceeded. If any comparison field entries are found that are less than the key A, then the Table Entry pointer, relative to the same base as the resolved B address, (for the first field that is less than or equal to all those fields that are less than the key), is stored in IX1 with the same Base Indicant as that of the resolved B operand.

If no entries are found that are less then the key A, then the null list value "CiEEEEEE" is stored in IX1.

Search Highest

The Search Highest search is only terminated when the Table Limit is exceeded. If any comparison field entries are found that are greater than the key (A), then the Table Entry pointer relative to the same base as the resolved B address, (for the first field that is greater than or equal to all those fields that are greater than the key), is stored in IX1 with the same Base Indicant as the resolved B address.

If no entries are found that are greater than the key A, then the null list value "CiEEEEEE" is stored in IX1.

No Bit Equal

All of the key field A must be logically ANDed with all of the comparison field. If the result is equal to zero, a match was found.

Comparison Flags

If the comparison condition is met on the first comparison, the Comparison Flags are set to LOW. If the comparison condition is met on other than the first comparison, the Comparison Flags are set to EQUAL. If the comparison condition is not met at all, the Comparison Flags are set to HIGH.

VIRTUAL ENTER (VEN) OP = 35

Function

VEN allows a user to change addressing environments. The first operand tells the processor where the parameters are (in a data memory area). The second operand specifies the environment to enter. The instruction then loads the new MAT.

There are local and non-local VENs. A local VEN does not require that the new MAT be loaded, while a non-local VEN does require switching to a new MAT and loading it.

Format

35 AFBF A B

AFBF Field

The AFBF field contains the length, in bytes, of the parameter field. The maximum number of bytes moved is 9,999, and a value of 0000 indicates that no data is to be moved. AF literals of B1, B2, or B3 are interpreted as lengths of 1, 2, or 3 characters in the A field, respectively. All other literal values cause an Invalid Instruction fault (IEX=22).

A Field

The A field contains the address of the parameter data field operand. This address can be indexed, indirect, or extended. The final address controller must equal UA or an Invalid Instruction fault (IEX=03) will be caused.

B Field

The B field contains the address of the 20-digit Environment field. This address can be indexed, indirect, or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) will be caused.

The 20-digit Environment field is defined as follows.

INFORMATION	DIGITS
Environment Number (EN)	00 - 05
Branch Address	06 - 11
Reserved	12 - 19

IMPLEMENTATION

This instruction checks the stack limit, stores control information and parameters onto the user's stack, and then executes an unconditional branch to a location specified by the Environment field, B. These steps are discussed below.

First, the Environment field B is checked for either of the following cases.

- If the Reserved area is not equal to zero, an Invalid Instruction fault (IEX=06) is caused and the instruction is terminated.
- If the EN is equal to zero, zeros will be stored in the Active Environment Number field of the Virtual Enter Stack Frame. (Refer to the Virtual Enter Stack Frame layout that follows.)

The Top of Stack (TOS) pointer is located at memory address 40, relative to Base #0. It is used as the atarting address for storing the Virtual Enter Stack Frame.

The sum of the TOS pointer, the size of the Virtual Enter Stack Frame (30), the amount of parameters (AFBF x 2), and the size of the Hardware Call Stack Frame (500) is compared to Limit #0. If the sum is greater than or equal to Limit #0, then a Stack Overflow fault will be incurred, and the instruction will be terminated. Otherwise, the Virtual Enter Stack Frame will be stored in the following order.

	INFORMATION	DIGITS
Old TOS —->	Measurement Register User)	00 - 05
	COM and OVF Flags	06 - 07
	Active Environment Number	08 - 13
New IX3 —->	Next Instruction Address	14 - 19
	Saved IX3 Value	20 - 27
	Stack Frame Indicator (FF)	28 - 29
	Stack Parameters (0-9,999 bytes)...	
New TOS —->		

The COM and OVF Flags contain the following information in the least significant digits. The other digit is reserved for future use, and currently equals zero.

INFORMATION	BIT
Reserved	3
Overflow Flag	2
Comparison LOW Flag	1
Comparison HIGH Flag	0

The parameters are moved from memory location A to the stack.

IX3 is then set. The two most significant digits are set to the value "C0". The six least significant digits are set to the initial address specified at memory location 40+14, relative to Base #0. Therefore, these digits contain a pointer to the Next Instruction field of the Virtual Enter Stack Frame.

The new value of the next available stack location, relative to Base #0, is stored into memory location 40, relative to Base #0.

The COM and OVF flags are then reset. (The Mode Indicators, Accumulator, Measurement register, Mobile Index registers, and the Interrupt Mask are not altered by this instruction.)

Local and Non-Local VENs

If the EN is equal to zero, then this instruction is a local VEN and an environment change will not be necessary. The correct Base/Limit pairs will already be resident in the processor. Therefore, the Active Environment Number will remain unchanged.

In the case of a non-local VEN, the environment must be changed. EN will not be equal to zero and this EN replaces the Active Environment Number. The new MAT specified by the new Active Environment Number must be located and loaded.

Finally, this instruction executes an unconditional branch to the address, relative to Base #1, that is contained in the Branch Address portion of the Environment field B.

It is important that the Active Environment Table being entered share the same Data Area (Base #0), for the processor will not check for improper memory assignments.

WRITE HARDWARE REGISTER (WHR) OP = 65

Function

This privileged instruction is used by the MCP to establish the absolute memory addresses of the Reinstate List, Snap Picture, Memory Error Report, or the Memory Area Status Table.

Format

65 AF BF A

AF Field

The AF field is unused and reserved, but can be specified as an indirect field length. A literal flag will cause an Invalid Instruction fault (IEX=21).

BF Field

The BF field is a variant and can be specified as an indirect field length. The following variations can be specified by this field after any indirect field length has been resolved. The use of any other BF values than those listed below is reserved and will cause an Invalid Instruction fault (IEX=26).

CASE	BF
Reinstate List Address	00
Snap Picture Address	01
Memory Error Report Address	02
Memory Area Status Table Address	03

A Field

The A field contains the address of the data field in memory. This address can be indexed, indirect, or extended. The final address controller must equal UN or an Invalid Instruction fault (IEX=03) will be caused.

If any of the address values in the A field are invalid (undigits), an Address Error fault (AEX=34) will be caused.

IMPLEMENTATION

The Case Where BF = 00 (REINSTATE LIST ADDRESS)

The A operand is used to locate the 9-digit absolute memory address of the Reinstate List pointer. Any references based on this pointer must be recalculated.

The Case Where BF = 01 (SNAP PICTURE ADDRESS)

The A operand is used to locate the 9-digit absolute memory address of the Snap Picture. Snap Picture Enable will be set to the value one.

The Case Where BF = 02 (MEMORY ERROR REPORT ADDRESS)

The A operand is used to locate the 9-digit absolute memory address of the Memory Error Report. Memory Error Report Enable will be set to the value one and Memory Error Report Pending will be set to the value zero.

The Case Where BF = 03 (MEMORY AREA STATUS TABLE ADDRESS)

The A operand is used to locate the 9-digit absolute memory address of the Memory Area Status Table pointer. Any references based on this pointer must be recalculated.

INDEX