B 1000 SYSTEM SOFTWARE RELEASE MARK 10.0

DOCUMENT/MCPMANUAL

B 1000 MCPII

REFERENCE DOCUMENT

TABLE OF CONTENTS

INTRODUCTION 1-1
RELATED DOCUMENTATION 1-1
S-MACHINE 1-2
SOFTWARE 1-3
FIRMWARE 1-3
TERMINOLOGY AND DEFINITIONS 2-1
MEMORY MANAGEMENT AND MEMORY LINKS 2-1
SEGMENT DICTIONARIES AND SYSTEM DESCRIPTORS 2-3
INTERPRETER MANAGEMENT, PARAMETER BLOCKS AND DICTIONARIES 2-5
CODE FILES, PROGRAM PARAMETER BLOCKS AND FILE PARAMETER BL 2-8
FILE INFORMATION BLOCKS 2-9
RUN STRUCTURE 2-10
RUN STRUCTURE NUCLEUS 2-11
DATA AND FILE DICTIONARIES 2-12
RE-ENTRANT PROCESSING AND CODE SEGMENT DICTIONARIES . . 2-12
THE I/O SUBSYSTEM 3-1
I/O DESCRIPTORS 3-2
GISMO - THE I/O DRIVER 3-4
CHANNEL TABLE 3-6
GISMO/HARDWARE INTERFACE 3-7
CA/RC CYCLES 3-8
PROCESSOR I/O INSTRUCTIONS 3-8
SERVICE REQUEST 3-9
STATUS COUNTS 3-10
DATA TRANSFERS 3-12
I/O CHAINING 3-13
DISK I/O CHAINING 3-14
DISK I/O OVERLAPPED SEEKS 3-15
TAPE I/O CHAINING 3-16
MONITORING OF PERIPHERAL STATUS 3-18
I/O ASSIGNMENT TABLE 3-18
UNIT MNEMONICS 3-22
TEST.AND.WAIT I/O OPERATORS 3-23
STATUS PROCEDURE 3-23
DISK IDENTIFICATION - PACK LABELS 3-24
PACK INFORMATION TABLE 3-25
TAPE LABELLING, INITIALIZATION AND PURGING 3-26
PE/NRZ EXCHANGES 3-31
FILE STRUCTURES 3-33
CONVENTIONAL FILES 3-33
FILE NAMING CONVENTIONS 3-35
LOGICAL DISK FILES 3-36
PHYSICAL DISK FILES 3-36
DISK SPACE ALLOCATION 3-36
FILE ACCESS AND IDENTIFICATION 3-37
DISK FILE IDENTIFICATION 3-38
MULTI-PACK FILES 3-39
BASE PACKS 3-40

CONTINUATION PACKS 3-40
MULTI-PACK FILE INFORMATION TABLE 3-41
MULTI-PACK FILE GENERAL RESTRICTIONS 3-42
PRINTER FILES 3-43
LINAGE Clause 3-45
PRINTER AND PUNCH BACKUP CAPABILITIES 3-48
BACKUP FILE BLOCKING FACTORS 3-49
BACKUP FILE CONTROL INFORMATION 3-51
BACKUP FILE LOGICAL RECORD FORMAT 3-52
Relative Files 3-54
Direct Files 3-54
Relative File Data Structure 3-54
Relative File Disk Initialization 3-55
Relative File Parameter Blocks (FPBs) 3-55
Relative Disk File Headers (DFHs) 3-55
Relative File Information Blocks (FIBs) 3-56
Relative File Communicate Operators 3-56
Indexed Sequential Files 3-58
Direct Files 3-58
Index Files 3-58
Cluster Files 3-59
Indexed Sequential Data File Structure 3-61
Indexed Sequential Index File Structure 3-62
Indexed Sequential Memory Structures 3-66
FIB Dictionaries 3-67
Indexed Sequential User Specific Information (USI) 3-68
Indexed Sequential File Global Information (GLOBALS) . 3-69
Disk File Header Extensions 3-72
Indexed Sequential Disk File Header Extension 3-72
Indexed Sequential Available Space Allocation 3-72
Index File Table Splitting 3-75
Current Record Pointer (CURRENT) 3-75
CURRENT Maintenance 3-76
Indexed Sequential Buffer Management 3-77
Indexed Sequential Buffer Descriptor (BD) 3-78
Concurrent Update Operations 3-79
Disk I/O Error Procedures 3-79
The Offset Procedure 3-80
The Strobe Procedure 3-81
The Error Correction Procedure 3-82
Data and Address Error Recovery - 215 And 225 Drives . 3-82
Data and Address Error Recovery - 205 And 206 Drives 3-83
Data and Address Error Recovery - 207 Drives 3-83
Data and Address Error Recovery - Disk Cartridges 3-84
Remainder of the Disk I/O Error Procedure 3-84
Tape I/O Error Procedures 3-85
S-MEMORY MANAGEMENT AND MEMORY REQUIREMENTS 4-1
GENERAL MEMORY MANAGEMENT CONCEPTS 4-1
LINKED MEMORY 4-1
TYPES OF MEMORY REQUESTS 4-2
THE FENCE 4-3
MINIMIZATION OF "CHECKERBOARDING" 4-4
VICTIM SELECTION 4-4
ROUND-ROBIN VICTIM SELECTION 4-5

WORKING SET DETERMINATION 4-6
SECOND CHANCE VICTIM SELECTION 4-6
PRIORITY VICTIM SELECTION 4-7
PROGRAMMATIC DETECTION OF MEMORY THRASHING 4-9
MEMORY INITIALIZATION 4-10
MEMORY REQUIREMENTS 4-15
OPERATING SYSTEM STATIC REQUIREMENTS 4-15
OPERATING SYSTEM DYNAMIC REQUIREMENTS 4-21
PROGRAM-DEPENDENT STATIC REQUIREMENTS 4-26
PROGRAM-DEPENDENT DYNAMIC REQUIREMENTS 4-28
M-MEMORY MANAGEMENT 5-1
DISTRIBUTION 5-1
CONTENTION 5-1
PROCESS (PROGRAM) MANAGEMENT 6-1
DEMAND MANAGEMENT 7-1
MCP OUTER LOOP 7-1
TIMER INTERRUPT 7-1
I/O INTERRUPTS 7-1
JOB SCHEDULING AND INITIALIZATION 7-2
COMMUNICATES 7-3
PROGRAM REINSTATE 7-4
PROGRAM COMMUNICATES 7-4
COMMUNICATE FORMAT 7-5
READ (MICRO MCP) 7-6
WRITE (MICRO MCP) 7-9
SEEK (MICRO MCP) 7-12
SORTER CONTROL 7-13
SORTER READ (MICRO MCP) 7-15
OPEN (DM) 7-16
CLOSE (DM) 7-17
OPEN 7-17
Disk File OPEN 7-28
CLOSE 7-32
POSITION (MICRO MCP (BACKUP FILES ONLY)) 7-38
ACCESS FILE PARAMETER BLOCK (FPB) 7-41
ACCESS FILE INFORMATION BLOCK (FIB) 7-42
DATA OVERLAY 7-42
ACCESS DISK FILE HEADER (DFH) 7-43
FIND/MODIFY (DM) 7-45
STORE (DM) 7-45
DELETE (DM) 7-46
CREATE/RECREATE (DM) 7-46
SWITCH.TAPE.DIRECTION 7-47
TERMINATE (STOP RUN) 7-47
FREE (DM) 7-50
TIME/DATE/DAY 7-50
INITIALIZER I/O 7-51
WAIT (SNOOZE) 7-52
ZIP 7-52
ACCEPT 7-53
DISPLAY 7-53
USE/RETURN 7-54
SORT HANDLER 7-54
SDL TRACE 7-55

ENULATOR TAPE (MICRO MCP) 7-55
COBOL PROGRAM ABNORMAL END 7-57
SORT EOJ 7-58
FREEZE/THAW RUN STRUCTURE 7-58
COMPILE CARD INFORMATION 7-58
DYNAMIC MEMORY BASE 7-59
MEMORY DUMP TO DISK 7-59
GET SESSION NUMBER 7-61
DC.INITIATE.IO 7-61
NDL/MACRO COMMUNICATES 7-62
DCWRITE 7-62
QUICK QUEUE WRITE (REMOTE FILES) 7-62
QUICK QUEUE WRITE (STATION NUMBER) 7-62
ACCESS USERCODE FILE 7-62
PROGRAM CALLER 7-63
LOAD.DUMP MESSAGE 7-64
COMPLEX WAIT (MICRO MCP) 7-64
MESSAGE COUNT 7-65
RECOVERY COMPLETE 7-65
GET.ATTRIBUTES 7-65
CHANGE.ATTRIBUTES 7-66
ACCESS.GLOBALS 7-66
INDEXED SEQUENTIAL POSITION 7-67
INDEXED SEQUENTIAL READ 7-68
INDEXED SEQUENTIAL WRITE 7-69
INDEXED SEQUENTIAL REWRITE 7-69
INDEXED SEQUENTIAL DELETE 7-69
RELATIVE I/O COMMUNICATE - START 7-70
RELATIVE I/O COMMUNICATE - WRITE 7-70
RELATIVE I/O COMMUNICATE - REWRITE 7-71
RELATIVE I/O COMMUNICATE - DELETE 7-71
RELATIVE I/O COMMUNICATE - READ 7-72
SEQUENTIAL REWRITE (MNCP) 7-72
INDEXED/SEQUENTIAL OPEN 7-73
INTER-PROCESS COMMUNICATION 8-1
QUEUE SYSTEM AND INTERFACES 8-1
DESIGN PHILOSOPHY 8-1
QUEUE FILE FAMILIES 8-1
QUEUE DESCRIPTORS 8-2
QUEUE DISK 8-2
MESSAGE DESCRIPTORS 8-3
MESSAGE BUFFERS 8-3
QUEUE ATTRIBUTES 8-4
QUEUE FILE LOGICAL I/O OPERATIONS 8-5
WRITING TO THE TOP OF A QUEUE FILE 8-7
MESSAGE.COUNT COMMUNICATE 8-7
INTER-PROGRAM COMMUNICATION 8-10
RUN UNIT DEFINITION 8-10
IPC IMPLEMENTATION OF SHARED DATA 8-11
IPC RUN STRUCTURE NUCLEUS CHANGES 8-12
RS.RUN.UNIT BIT(16) 8-12
RS.RUN.UNIT.LINK BIT(16) 8-12
RS.IPC.DICT BIT(24) 8-12
RS.IPC.PARAMETER.LIST BIT(24) 8-13

RS.IPC.DICT.SIZE BIT(16) 8-13
RS.EXECUTE.TYPE BIT(4) 8-13
RS.NAME CHARACTER(30) 8-13
RS.CALLERS.LR BIT(24) 8-14
RS.IPC.EVENT BIT (1) 8-14
RS.CANCELED BIT(1) 8-14
IPC Program Parameter Block Changes 8-14
PROG.IPC.SIZE BIT(16) 8-14
PROG.IPC.PTR BIT(24) 8-15
PROG.IPC.MAX.SEND.PARAMS BIT(16) 8-15
IPC.DICTIONARY 8-15
IPC COMMUNICATE OPERATOR 8-16
IPC Verb Operation 8-16
IPC CALL OPERATION 8-17
IPC CANCEL OPERATION 8-19
IPC EXIT PROGRAM OPERATION 8-19
IPC TERMINATION CONSIDERATIONS 8-19
IPC MICRO MCP/S.MCP COMMUNICATION 8-20
IPC PROGRAM DUMPS 8-21
IPC CANDIDATES FOR ROLL-OUT 8-21
IPC TASK CONSIDERATIONS 8-21
IPC PROGRAM NAME SPECIFICATIONS 8-21

INTRODUCTION

The purpose of this document is to define and discuss the Master Control Program II (MCP) for the B1000 machines. The concept and design of the MCP will be discussed and the functional specifications of the MCP's operations will be catalogued.

The sort, data communication, and data management systems will not be discussed in any depth in this document. Detailed descriptions of these features appear in other Burroughs publications (See Related Documentation below).

RELATED DOCUMENTATION

Name	Number
B1000 MCP Utilities	P.S. 2212 5579
B1000 Network Definition Language	P.S. 2212 5223
B1000 Data Management Systems II	P.S. 2212 5470
B1800/B1700 Sort	P.S. 2201 6752
B1000 Software Operational Guide	1068731

These specifications are written for those people with programming experience and a knowledge of basic software concepts. Those unfamiliar with operating system design will gain insight into the Burroughs philosophy of system management. Those individuals familiar with operating systems of other manufacturers or of other Burroughs machines will gain an understanding of the Master Control Program implemented specifically for the Burroughs B1000.

Also included in this specification are brief descriptions of various functions performed by the micro-coded I/O driver routines. These same routines are often referred to as "GISMO" and "I/O interpreter". The discussions are necessary for completeness and for a thorough understanding of the B1000 operating system of which the I/O driver is an integral part.

MCP II is a modular, supervisory program that assumes common, logically complex functions to simplify and expedite the tasks of programming and system operation. Its most important duties include such functions as:

* Scheduling, initiation, running, and termination of jobs

* Providing a symbolic means of communicating with the system while shielding the user from the detail of the hardware

* Providing a family of common facilities such as management of input/output operations and file maintenance

* Managing the system's resources for optimum utilization in a multi-programming environment

S-MACHINE

The B1000 is a small-to-medium scale, general purpose computer system. Its distinguishing feature is its flexibility, made possible through interpretive processing. In any computer system a representation of any process has two components: (1) a family of structures representing the state of that process, and (2) a series of operators able to manipulate those structures. Until the advent of fourth generation computers, both components were represented in the machine hardware itself. A compiler or language translator transformed the source code (e.g., COBOL, FORTRAN) into a "machine language" (object code) which was defined in terms of the hardware architecture.

For the set of processes able to be generated by any particular programming language, there exists a machine architecture which best represents those processes. For instance, COBOL is a character-oriented language and performs decimal arithmetic exclusively. Because of its data manipulation features, it might best utilize a machine architecture with multi-address operators, capable of performing efficient "moves," "compares," and simple expression evaluation. On the other hand, FORTRAN was designed to compute complex mathematical functions. It favors a stack structure for parameter passing and complex expression evaluation. It performs binary arithmetic and would prefer 30- to 50-bit word sizes.

The difficulty of designing a hardware structure capable of handling two such divergent languages in the most efficient manner becomes apparent. It would be possible, in principle at least, to design the hardware in such a way as to adequately represent both sets of structures. However, this would prove to be prohibitively expensive. The typical approach, therefore, has been to either design the hardware to favor one language at the expense of others or to design a compromise structure capable of handling several languages, but none in the most efficient manner. The wide variety of programming languages in current use has placed a great strain on the capacity of the hardware to efficiently execute code compiled from very different languages.

It is to this problem that designers of fourth generation software, and the B1000 in particular, have addressed themselves. Rather than build a particular structure into the hardware, the concept of the "soft machine" has been developed whereby the ideal environment of structures and operators is programmatically simulated.

The B1000 hardware was designed with as little explicit structure as possible. Because memory may be addressed to the bit, no one structure is inherently favored over any other. The only required structure is that which will allow the simulation of any "soft machine". Thus the range of structures able to be represented on the B1000 is unlimited.

As stated previously, for every compiler language there exists a machine architecture within which the algorithms generated by that compiler will best run. On the B1000 this hypothetical environment is called the "S-machine". An S-machine has been defined for each language such that any process may be represented in its most efficient or most natural form, unrestrained by any arbitrary hardware configuration.

Compilers on the B1000 generate code files which contain (1) the information necessary to initialize the appropriate S-machine at run time, and (2) the "S-code" to be executed on this S-machine. S-code is written in S-language, the machine language for an S-machine. Execution is achieved by the S-code being interpreted, an S-operator at a time, by a micro-program called an interpreter.

SOFTWARE

The term "software", as used in this document, refers to all programming supplied by the Santa Barbara Plant. When the term is used, it most likely is referring to programs that are written in a higher-level language. This may not always be the case, but typically, the term will refer to the compilers and utility programs created by the Programming Activity.

FIRMWARE

The firmware consists of a set of interpreters, those portions of the MCP which are micro-coded and reside in an entity known as the MICRO/MCP, and a program called "GISMO". For each S-language a micro-coded program called an interpreter acts upon the hardware and executes the compiled S-code as defined by the S-machine. The B1000 software has been implemented in such a way

that any number of interpretive structures may be active in the system at any given time. This is achieved by dynamically establishing, upon demand, the S-machine structure for any process.

For instance, the MCP, which is itself a program, is written in a high-level language, SDL, that is designed specifically for writing software. It has its own optimum environment (the SDL S-machine) consisting of the structures and operators required for software applications. It has its own S-language and its own interpreter (the SDL interpreter). Running simultaneously in the system may be another program written in a different language (e.g., COBOL). This program also has its own structure (the COBOL S-machine), S-language, and interpreter. The system, when executing the MCP's supervisory functions, assumes the architecture of the SDL S-machine and, when executing the COBOL instructions, takes on the COBOL S-machine structure. This switching of interpreters and process environments is managed completely by the software and is invisible to the user of the machine.

The B1000 MCP has actually evolved to its present state. Originally, all functions of the MCP were coded in SDL. Beginning with the 4.0 release of the software, the most commonly used routines of the MCP were written in micro-code and placed in GISMO. This resulted in substantial performance improvements. Beginning with the 5.1 release of the software, these commonly used routines were removed from GISMO and placed in the entity mentioned previously, the MICRO/MCP.

These specifications have also evolved along with the MCP. Many of the functions described herein are now performed by the MICRO-MCP, though the function itself remains exactly the same as it was when it was performed by SDL code. Since this document is intended to be a functional specification of the B1000 operating system, all MCP functions are described herein. Whether the function is performed by SDL code or by micro code should be completely transparent to the user. Actually, the functional result is the same for both, but the time and resource requirements are not identical. The difference is therefore not always transparent.

Throughout this document, the acronym "MCP" may be referring to the MICRO/MCP or to the SDL MCP. In cases where the distinction is important, "MCP" will not be used but the two terms mentioned above will be. This document, then, will actually be a functional specification of the operating system, as it was originally intended to be, though it will actually be describing two separate and distinct programs. Since GISMO is also a critical part of the operating system, the document may also

touch upon portions of GISMO.

GISMO is a micro-coded family of critical routines common to all processes. GISMO may also be referred to in this document as "CSM", an acronym for Central Service Module. It is a central module of service routines used by all programs in the system and performs three basic functions:

1. Switching of control between all contending processes in the system.

2. Recognition and queueing of interrupts received from the I/O controls or from other processes in the system.

3. Initiation and management of the I/O controls connected to the machines, usually at the request of another process.

Processor allocation, the switching of control between two or more processes, is handled by the "Micro Scheduler" module in GISMO. This module may be thought of as an "Outer Loop". It has absolute control over the process which will be performed next on the system.

Interrupt resolution consists of routines which perform certain functions depending on the type of interrupt and certain other critical conditions. The interpreter in control senses the interrupt and calls upon GISMO to take the required action.

GISMO's service request module (soft I/O) performs the function of a hardware device capable of performing a memory access at the request of an I/O control. An I/O control on the B1000 is a hardware device which acts as an interface between soft I/O and a peripheral device. It requests access to memory on behalf of the device and manages the device itself. The collection of I/O controls is called the I/O sub-system.

Typical data transfer operations involve frequent but brief calls upon soft I/O by the I/O sub-system. The firmware was designed in such a way that between the execution of any two S-operators, the interpreter in control will check a flag in the processor (called the Service Request Bit) to see if the I/O sub-system is demanding attention. If it is, the interpreter passes control to GISMO which performs the necessary memory access and returns control to the interpreter.

TERMINOLOGY AND DEFINITIONS

Before proceeding with a detailed description of what the MCP does and how it goes about it, it will be necessary to define a number of terms and data structures whose names are used familiarly throughout the document. The reader should know the meanings of the terms, but a thorough understanding of the many diverse programming structures presented herein is not required. The structures are presented only in the interests of completeness, and as a possible aid in understanding the narrative descriptions of the MCP's functions, presented in the later sections of the specification.

MEMORY MANAGEMENT AND MEMORY LINKS

The MCP organizes and allocates space in memory through the use of fields known as memory links. Each link immediately precedes the block of memory it describes and includes such information as: The size of that block of memory; the type of use (if any) to which it is put; and pointers to the immediately preceding and succeeding links. If the block of memory is classified as available (i.e., not currently in use by any process), an additional set of descriptors point to the links of the prior available and next available blocks of memory. Thus it is possible to search all links or only those links describing available memory. A programmatic description is given below:

```
DEFINE  MEMORY.LINK.SIZE AS #187#;
DECLARE MEMORY.LINK TEMPLATE BIT(MEMORY.LINK.SIZE);
DEFINE MEMORY.LINK.DECLARATION AS #
DECLARE 01 DUMMY REMAPS MEMORY.LINK,
           2 ML.DISK                      DSK.ADR,
           2 ML.GROUP,
             3 ML.POINTER                 ADDRESS,
             3 ML.JOB.NUMBER              BIT(16),
             3 ML.TYPE                    BIT(6),
             3 ML.SAVE                    BIT(1),
           2 ML.SIZE                      BIT(24),
           2 ML.PRIORITY.FIELD            BIT(30),
             3 ML.DK.INTERVAL             BIT(10),
             3 ML.CURRENT.DK.INT          BIT(10),
             3 ML.INCOMING.PRIORITY       BIT(5),
             3 ML.RESIDENCE.PRIORITY      BIT(5),
               4 ML.RP.WHOLE              BIT(4),
               4 ML.RP.FRACTION           BIT(1),
           2 ML.FRONT                     BIT(24),
           2 ML.BACK                      BIT(24),
           2 ML.USAGE.BITS                BIT(2),
             3 ML.PREVIOUS.SCAN.TOUCH     BIT(1),
             3 ML.CURRENT.SCAN.TOUCH      BIT(1);#;
```

```
USE( ML.DISK
     ,ML.POINTER
     ,ML.JOB.NUMBER
     ,ML.TYPE
     ,ML.SAVE
     ,ML.SIZE
     ,ML.FRONT
     ,ML.BACK
      ) OF MEMORY.LINK.DECLARATION;

DEFINE Q.ML.DECLARATION AS#DECLARE
   01 Q.MEMORY.LINK TEMPLATE
,     02 FILLER                BIT(MEMORY.LINK.SIZE)
,     02 Q.ML.F.AVL            ADDRESS
,     02 Q.ML.B.AVL            ADDRESS
;#;

DEFINE
   TAKE.LO                     AS#0#
,  TAKE.RIGHTMOST              AS#1#
;
DEFINE          % TYPES FOR "ML.TYPE"
     CODE AS          #0#%

,    AVAILABLE AS     #2#%
,    RN.S AS          #3#%
,    MCP.TEMP AS      #4#%
,    USER.FILE AS     #5#%
,    SEG.DICTV AS     #6#%
,    MICROCODE        #7#%
,    DICT.MASTER AS #8#
,    QUEUE.DIRECTORY.TYPE
                   AS #9#
,    MSG.BUFFERV
                  AS  #10#
,    MESSAGE.LIST.TYPE
                   AS #11#
,    TO.BE.FORGOTTEN AS #12#
,    DATA.SEG     AS #13#
,    DBM.BUFFER AS  #14#
     TERMINATING LINK AS #15#%
,    MCP.PERM     AS #16#%
,    PSR.MEM      AS #17#%
,    MCP.IOAT     AS #18#%
,    DISK.HEADER AS #19#%
,    PACK.MEM     AS #20#%
,    SD.CNTNR     AS #21#%
,    SCHED.MEM    AS #22#%
,    SORT.MEM     AS #23#%
,    DCH.MEM      AS #24#%
,    MICROCODE.NON.OVERLAYABLE AS #25#%
,    QUEUE.AVL.BUF.V AS #26#
,    DMS.DISK.HDR AS#27#%
```

```
,   DMS.STRUCTURE AS#28#Z
,   DMS.TEMP      AS #29#Z
,   DMS.GLOBALS AS #30#Z
,   DMS.TEMP.LOCK.DESCR AS #31#
,   XM.MEMORY AS #32#
,   PERM.SPO.BUFF AS #33#
;
```

"TEMPLATE" in the above description is defined as "REMAPS BASE". This is not important to an understanding of memory link operation. "ADDRESS" is defined in the MCP symbolic as "BIT(24)". The word "ADDRESS" here is used as a denotation of memory address. Hence, "ML.BACK" in the description above is a pointer to the previous memory link and "ML.FRONT" is a pointer to the succeeding link. ML.SIZE will contain the size of the area, in bits, and ML.GROUP is valid only if the area is in use. ML.POINTER will contain the memory address of the segment dictionary entry associated with this memory area. Segment dictionaries are described in the next section. ML.JOB.NUMBER will contain the job number of the program using the area. ML.SAVE, the description of which is defined as "BOOLEAN," is set on if the memory area must be saved on disk before it is overlaid.

As can be determined by adding the sizes of the various components, a memory link requires 187 bits of storage space. Since memory is allocated dynamically, it is often difficult to predict with any degree of accuracy exactly how much memory will be required by any task. The sizes of all memory links involved must be included in the calculations. This is discussed further in a later paragraph.

SEGMENT DICTIONARIES AND SYSTEM DESCRIPTORS

Virtual memory is supported by allowing process segmentation. By segmenting code, data, and interpreters and dynamically moving a segment into or out of memory as required, the system is able to function as if it had "virtually infinite" memory capacity. The MCP manages this facility through three structures: Code Segment Dictionaries, Data Segment Dictionaries, and Interpreter Segment Dictionaries. Each dictionary consists of a string of system descriptors each of which describes one segment including its length, location and status. As a segment is moved in or out of memory its dictionary entry is updated accordingly.

At run time the MCP creates the code and data segment dictionaries from information in the program's code file. The interpreter segment dictionary is created from the interpreter code file in the same manner and is referenced by an entry in the

interpreter dictionary, a structure fixed in memory at Clear/Start time. The run structure of the program contains pointers to the code and data segment dictionaries and an index into the interpreter dictionary. A programmatic description is given below:

SYSTEM DESCRIPTORS

```
DECLARE
    01 SYSTEM.DESCRIPTOR TEMPLATE BIT(SY.SIZE);
 %
DEFINE SY.DECLARATION AS #SY.DECL(SYSTEM.DESCRIPTOR)#;%
DEFINE SY.DECL(X) AS #DECLARE%
    01  DUMMY REMAPS X,%
        02 SY.IN.USE            BIT(1),   % TO HELP MEMORY MANAGEMENT
        02 SY.MEDIA             BIT(1),   % 0=DISK, 1=S-MEMORY
        02 SY.LOCK              BIT(1),   %
        02 SY.IN.PROCESS        BIT(1),   % TRUE IF THERE IS AN I/O IN
                                          % PROCESS FOR THE INFORMATION
                                          % REPRESENTED BY THIS DESCRIPTOR.
                                          % IF TRUE, "SY.CORE" CONTAINS A
                                          % POINTER TO THE I/O DESCRIPTOR.
        02 SY.INITIAL           BIT(1),   % "ADDRESS" IS READ-ONLY MOTHER
                                          % COPY, HENCE IF "WRITE" THEN GET
                                          % NEW DISK AND REPLACE ADDRESS.
        02 SY.FILE              BIT(1),   % THE OBJECT OF THIS DESCRIPTOR
                                          % IS A FILE WHOSE USERCOUNT MUST
                                          % BE DECREMENTED WHEN THIS
                                          % DESCRIPTOR IS RETIRED.
        02 SY.DK.FACTOR         BIT(3)    % MEMORY DECAY FACTOR
        02 SY.SEG.PG            BIT(7),   % MEMORY.ACTIVITY AUDITING
        02 SY.TYPE              BIT(4),   % UNITS FOR SY.LENGTH.
                                          % 0 = BITS
                                          % 1 = DIGITS (4 BIT)
                                          % 2 = CHARACTERS (8 BIT)
                                          % 3 = NORMAL DESCRIPTORS
                                          % 4 = DISK SEGMENTS
                                          % 5 = SYSTEM DESCRIPTORS
                                          % 6 = SYSTEM INTRINSIC
                                          % 7 = INDIRECT REFERENCE
                                          %     ADDRESS GIVES RELATIVE
                                          %     DISPLACEMENT IN BITS
                                          %     (SIGNED NUMBER).
                                          % 8=  MICROS
        02 SY.ADDRESS           BIT(36),  %
           03 FILLER            BIT(12),  % PORT,CHANNEL AND UNIT.
           03 SY.CORE            BIT(24),  % CORE, OR ADDRESS WITHIN UNIT.
        02 SY.LENGTH            BIT(24);  % NUMBER OF UNITS, AS DETERMINED
                                          % BY SY.TYPE.
#;                                        %
%
%
```

```
%
   DEFINE ND.DECLARATION AS#
   DECLARE
   01 DUMMY REMAPS NORMAL.DESCRIPTOR BIT(ND.SIZE),
      02 NO.DK.FACTOR       BIT(3),
      02 FILLER             BIT(6),
      02 ND.CORE            BIT(24),
      02 ND.TYPE            BIT(3),
      02 ND.LENGTH          BIT(24);#;
```

SY.SIZE is defined in the MCP code as eighty. Hence eighty bits are required to contain one segment dictionary entry, or system descriptor. The use of the term "DESCRIPTOR" in B1000 documentation is often misleading and ambiguous. There are many different types of descriptors, all of which have different memory requirements and formats. Consequently, system descriptors will always be referred to as such or as segment dictionary entries.

The comments on the various fields comprising the system descriptor are largely self-explanatory. Perhaps some explanation of selected fields would be beneficial, however. SY.LOCK is set true if the system descriptor describes a data field and if the interpreter is currently accessing the field. This is to avoid the situation which arises in a simple replacement statement where the sending and receiving field are both in overlayable segments. In order to do the replacement, both data segments must be in memory simultaneously.

SY.INITIAL is true for initialized data only. The most common case of this occurs when executing a COBOL program and the programmer has used the value clause to initialize data fields and the data field itself is in an overlayable segment. SY.ADDRESS may be either a disk or a memory address, depending on the setting of SY.MEDIA. If it is a memory address, the most significant twelve bits are ignored. If it is a disk address, the most significant twelve bits contain the port, channel and unit associated with the disk address.

INTERPRETER MANAGEMENT, PARAMETER BLOCKS AND DICTIONARIES

The B1000 MCP maintains a list or directory of all files on disk. file stored on disk has a unique name, which may consist of up to three fields, each of which may consist of up to ten characters. Associated with each file on disk is an item called a "Disk File Header". The disk file header serves essentially to describe the file. All of this is described in detail in later sections. This brief discussion is being included at this point to facilitate the following discussions on interpreters.

Included in the disk file header is a field which denotes the type of file. There are separate type numbers for data files, code files, interpreters, and so forth. Code files (programs) and interpreters are further described by the first disk segment contained in the file. This segment is called the "Program Parameter Block" or the "Interpreter Parameter Block", respectively. A detailed description of the program paramter block is presented in a later section. A programmatic description of the interpreter parameter block is presented below.

```
    DEFINE IPB.DECLARATION AS#
    DECLARE 01 DUMMY REMAPS IPB BIT(1440),
               02 FILLER BIT(1192),
               02 IPB.HARDWARE CHAR(1),
               02 IPB.ARCHITECTURE.NAME CHAR(10),
               02 IPB.COMPILER.LEVEL BIT(8),
               02 IPB.MCP.LEVEL BIT(8),
               02 IPB.GISMO.LEVEL BIT(8),
               02 IPB.ARCHITECTURE.ATTRIBUTES BIT(80),
               02 FILLER BIT(56);
#;
```

IPB.HARDWARE will contain either an "S" or an "M", depending upon whether the interpreter was generated for an S-memory or an M-memory processor. All B1800's are considered to be M-memory processors. IPB.ARCHITECTURE.NAME will contain the generic name of the compiler, such as COBOL or FORTRAN. IPB.COMPILER.LEVEL will be a number which will correspond to the release level of the software, as described below. IPB.MCP.LEVEL, IPB.GISMO.LEVEL and IPB.ARCHITECTURE.ATTRIBUTES are parts of the interpreter verification feature of the MCP.

The B1000 MCP includes facilities to recognize the hardware configuation it is executing upon and select the corresponding interpreter from the disk directory. All programs which are compiled for execution on a B1000 will have an interpreter "TYPE" requested the program parameter block of the code file (described in a later section), or the specific name of the interpreter to be used. As explained in a later section, the program parameter block contains space for three names to be associated with an interpreter. For discussion purposes here, the three names will be referred to as the "PACK" name, the "FAMILY" name and the "OFFSPRING" name.

The B1000 compilers generate the last two names of the interpreter only. The family name generated always corresponds to the language the program is written in, such as "COBOL" or

"FORTRAN". The offspring name is always one of the reserved words "INTERP", "DEBUG" or "TRACE". At BOJ, the MCP modifies the offspring name by concatenating one numeric character denoting the compiler level and either the character "M" or "S" depending upon whether the machine is equipped with an S-memory or an M-memory processor.

The level number concatenated is contained in the program parameter block as "PROG.COMPILER.LEVEL". Every time the compilers are changed in such a manner that the interpreter must also be changed, the level number generated by the compiler is incremented. The interpreters are then modified accordingly and released to the field under a new name. The new name will be the same as the old one, except for the level number contained in the name. For a COBOL program which is being executed on a B1720-series machine and had been compiled by the 4.1 COBOL compiler, the MCP will generate "COBOL"/"INTERP1M" as the interpreter name to be used for the execution. It should be noted that this feature was first included in the 4.1 software release. Level numbers were not included in the program parameter block prior to the 4.1 release.

Once the interpreter name is generated, the disk directory is searched for the interpreter. Upon finding the interpreter, the MCP will bring it into S-memory, if it is not already there, and construct an entry in the "INTERPRETER.DICTIONARY". All interpreters are re-entrant on the B1000. All of this is described in greater detail in the paragraph which follow. Each entry in the interpreter dictionary has the following format.

```
DEFINE ID.DECLARATION AS#DECLARE
    01 DUMMY REMAPS INTERPRETER.DICTIONARY,
       02 ID.SEG.DIC                    SY.DSCR,
       02 ID.ENTRY.IN.USE               BOOLEAN,
       02 ID.RSDNT.USERCOUNT            BIT(7),
       02 ID.TOTAL.USERCOUNT            BIT(7),
       02 ID.MIN.M.SIZE                 BIT(4),
       02 ID.MAX.M.SIZE                 BIT(4),
       02 ID.PARTIAL.BIT                BOOLEAN,
       02 ID.BLOCK.COUNT                BIT(4),
       02 FILLER                        BIT(19),
       02 ID.M.PRESENCE.BIT             BOOLEAN,
       02 ID.M.ADDR                     BIT(12),
       02 ID.TOPM                       BIT(4),
       02 ID.MEDIA                      BIT(2),
       02 ID.LOCK                       BOOLEAN,
       02 FILLER                        BIT (13),
       02 ID.TYPE                       BIT(4),
       02 ID.ADDRESS                    BIT(36),
          03 FILLER                     BIT(12),
          03 ID.CORE                    BIT(24),
```

02 ID.LENGTH BIT(24);#;

There is one entry in the interpreter dictionary for each interpreter presently in use. The I/O driver is always the first interpreter entered in the dictionary, the Micro MCP is the second entry and SDL is always the third entry in the dictionary. On the B1000, it is possible to segment interpreters. Consequently, a code segment dictionary is constructed for each interpreter as it is brought into memory. The system descriptor, the first item in the interpreter dictionary, is a pointer to the interpreter's code segment dictionary. Interpreters may be segmented, exactly as programs are. The same routines in the MCP are used for handling program segments and interpreter segments.

A certain amount of information about each program currently being executed is maintained in memory by the MCP. The field in which this information is maintained is known as the Run Structure Nucleus of the program. It is abbreviated as RS.NUCLEUS. In the RS.NUCLEUS, there is an index into the interpreter dictionary. All programs being executed at any given time which are using the same interpreter will have the same index in the field in their respective nucleus. In this manner, interpreter re-entrancy is accomplished.

The remaining field in the interpreter dictionary entry will not be described in detail at this point. For a more detailed description of interpreter management, the reader is referred to the section of this document which deals with M-memory management. It should be sufficient at this point to say that all interpreter segments except the first are treated as ordinary code and are considered overlayable. The first segment of each interpreter is not treated as code and is not overlayable, however.

The I/O driver, which is considered an interpreter, is an exception to the above statements.

CODE FILES, PROGRAM PARAMETER BLOCKS AND FILE PARAMETER BLOCKS

The code file of every program must contain two types of records to allow the MCP to manage the execution of that program: the "File Parameter Block" (FPB), and the "Program Parameter Block" (PPB). There is one FPB for each file declared in a program plus one entry for a trace file.

The first 2880 bits (two disk segments) of every code file is the "Program Parameter Block" (PPB) whose format is rigidly defined

by the MCP. Every compiler generates a PPB of the same format. It provides, for the MCP, all the vital statistics of the program including: The program's name; the name of the interpreter to be used during execution; the relative addresses of the FPB's, IPB, code segment dictionary and data segment dictionary, memory requirements for the program's execution; and tracing information.

At run time a working copy of the PPB is written into a temporary or permanent log (as dictated by the system options). The first two segments of this four segment entry are an exact copy of the PPB from the code file. Another segment is generated by the MCP and documents certain features of that particular execution. A final segment is reserved for an abnormal termination message.

If the code file is an interpreter code file, it contains an additional segment called the "Interpreter Parameter Block". It contains information concerning the software compatibility of the interpreter. A field in a program's PPB specifies under which interpreter it will run. When the program is scheduled for execution, the IPB of the interpreter named in the PPB is checked to insure that the interpreter is compatible with both the code file and the system software. The MCP informs the system operator via a SPO message if the interpreter cannot run. Refer to the appropriate MCP listing for a programmatic description.

The "File Parameter Block" (FPB) is a 1440-bit record created by the compiler from the user's file attribute declarations. Its format is rigidly defined by the MCP, and it contains the vital statistics which allow the MCP to manage the file's usage. When a job is scheduled for execution, a working copy of the FPB is written into a permanent or temporary log (depending on system options). In addition to recording the file's attributes, the MCP documents the use of the file during that job's execution. It records such information as the number of times the file was opened and closed; the total amount of time the file was open; the number of records read; the number of I/O errors; and the file type. Refer to the appropriate MCP listing for a programmatic description.

FILE INFORMATION BLOCKS

As each file is opened by the user program, a structure known as a File Information Block (FIB) is created in memory by the MCP. The FIB contains all information necessary for the MCP to perform normal, requested, I/O operations on the file. Much of the information in the FIB is taken directly from the FPB. Other information in the structure is inserted by the MCP, based upon the characteristics of the peripheral device assigned to the

file. Device assignment is discussed in the section of this specification which describes the Open Communicate.

FIB's vary in size, depending upon the type of device assigned to the file. Due to the amount of information which must be maintained, a disk file FIB is much larger than that of a card punch file, for example.

I/O descriptors and buffer memory areas are allocated and initialized by the MCP at the same time. There will therefore be one memory link only, for each file that is active in a program. Buffer areas and descriptors are not normally shared between files, though the Data Management subsystem, the Data Communications subsystem, the Relative file implementation and the Indexed file implemenation offer some exceptions to this rule.

A complete structural description of the FIB will not be presented herein, due primarily to the length of the structure. Also, the FIB is of interest to the various portions of the Operating System only. The programmatic description of the structure is readily available in the MCP listing. Sizes of FIB's for the different peripheral devices are presented in the following table.

File Assigned to:	Size in Bits
Reader-Sorter	742
Printer	724
Remote Device	557
Tape	724
Disk	976
Queue	385
All Other Devices	612

RUN STRUCTURE

The structure in memory that represents the state of any process is the run structure. Each process has a unique run structure. When a job is initialized before execution, the MCP creates the run structure from an analysis of the program's code file, and adds certain information it will need for management of the execution. All run structures are linked together by priority.

A run structure consists of a program"s data or address space, the MCP's managerial space called the run structure nucleus, and

the file and data segment dictionaries. The program's address space, residing between its base and limit registers, is that area of memory that may be accessed and manipulated by the program itself. A program's base register is a memory address that marks the lower bound of its addressable space. The limit register specifies the upperbound. A program may not access memory that is outside its own base to limit area, though this tenet is enforced by the interpreters and not the MCP.

A program's address space may contain both resident and overlayable data. The resident data area contains those fields which will be present in memory throughout the duration of the execution. The overlayable data space contains segmented data which may be brought into or out of memory as needed.

RUN STRUCTURE NUCLEUS

The Run Structure Nucleus is an area structured and maintained by the MCP and contains the essential information about the program. It resides in memory directly above the program's limit register and is accessible by the MCP and the program's interpreter. It contains such information as:

* Pointers to

  BASE AND LIMIT
  SEGMENT DICTIONARIES (CODE AND DATA)
  FILE DICTIONARY
  INTERPRETER DICTIONARY ENTRY
  NEXT RUN STRUCTURE (BY PRIORITY)
  CODE FILE ON DISK
  DISK LOCATION OF RUN STRUCTURE IF "ROLLED OUT"
  PROGRAM's LOG ENTRY
  VIRTUAL DATA SPACE ON DISK
  NEXT INSTRUCTION TO BE EXECUTED
  DMS POINTERS

* Structures necessary for communication between the program and the MCP

* Fields to reflect the state of the S-machine

* Fields for program switches

A programmatic description may be obtained from the MCP listing.

DATA AND FILE DICTIONARIES

The data segment dictionary resides at the end of the Run Structure Nucleus and is pointed to by a field in the nucleus. If there is no segmented data and the user has not requested that his resident data area be initialized, then the pointer will be null, and there will be no dictionary.

Each entry in the dictionary is an 80-bit system descriptor pointing to one data segment.

The last element of a run structure is the file dictionary. There is one 80-bit descriptor for each declared file plus one additional descriptor for a trace file (used for tracing). While a file is open, its dictionary entry points to the file's FIB in memory. If a file has never been opened, its entry is null. If a file has been temporarily closed (i.e., "CLOSE ROLLOUT"), its dictionary entry points to its FIB which has been written to disk. After a permanent close, the file's dictionary entry will again be null.

RE-ENTRANT PROCESSING AND CODE SEGMENT DICTIONARIES

The B1000 MCP allows re-entrant processing, the ability of two or more processes to use the same code segment dictionary and, thereby, the same code. The code segment(s) and code segment dictionary reside outside a program's run structure, and a field in the run structure nucleus points to its code segment dictionary. A structure called the segment dictionary container contains the information necessary to govern the use of a particular code segment dictionary. When a job is being initiated for execution, the MCP determines whether or not the code segment dictionary desired by the job is already in use. If it is, that dictionary will be used. The segment dictionary container reflects, among other things, the number of processes using the dictionary it describes. If there is more than one user, the segment dictionary container will remain in memory until all users have completed execution.

THE I/O SUBSYSTEM

This section of the specifications is a description of:

1. I/O Descriptors
2. GISMO Operation
   1. Channel Table
   2. GISMO/Hardware Interface
      1. CA/RC Cycles
      2. Processor I/O Instructions
      3. Service Request
      4. Status Counts
      5. Data Transfers
   3. I/O Chaining
   4. Disk I/O Chaining
   5. Disk I/O Overlapped Seek
   6. Tape I/O Chaining
3. Monitoring of Peripheral Status
   1. I/O Assignment Table
   2. Unit Mnemonics
   3. Test and Wait I/O Operators
   4. STATUS Procedure
   5. Disk Identification - Pack Labels
   6. Pack Information Table
   7. Tape Labelling, Initialization and Purging
   8. Tape PE/NRZ Exchanges
4. File Structures
   1. Conventional Files
      1. File Attributes
      2. File Naming Conventions
      3. Logical Disk Files
      4. Physical Disk Files
         1. Disk Space Allocation
         2. File Access and Identification
         3. Disk File Identification
         4. Disk File Header
      5. Multi-Pack Files
         1. Base Packs
         2. Continuation Packs
         3. Multi-Pack File Information Table
         4. Multi-Pack File General Restrictions
      6. Printer Files
         1. Logical/Physical I/O Relationship
         2. Logical Page Implementation
      7. Printer and Punch Backup Capabilities
         1. Backup File Blocking Factors
         2. Backup File Control Information
         3. Backup File Record Format
   2. Relative Files
      1. Direct Files
      2. Data Structure
      3. Disk Initialization

4. File Parameter Blocks
5. Disk Header
6. File Information Blocks
7. Communicate Operators

3. Indexed Sequential Files
    1. Direct Files
    2. Index Files
    3. Cluster Files
    4. Data File Structure
    5. Index File Structure
    6. Memory Structures
        1. FIB Dictionaries
        2. User Specific Information (USI)
        3. File Global Information (GLOBALS)
        4. Structure Descriptor
        5. Disk File Header Extension
    7. Available Space Allocation
    8. Index File Table Splitting
    9. Current Record Pointer
    10. Current Maintenance
    11. Buffer Management
    12. Buffer Descriptor
    13. Concurrent Update Operations

5. The I/O Error Procedures

There is some overlap between the information contained in this section of the specification and that contained in the Demand Management section of the document. The Demand Management section was originally intended to cover the management of the peripheral after it had been assigned to a user as a file; the I/O Subsytem section was intended to cover the management of the device up to that time. This division is not always possible, particularly in the case of disk devices. The reader may have to refer to both sections of the document to find the answer to a specific question.

I/O DESCRIPTORS

Normal state programs request I/O functions in a symbolic fashion (e.g., Write a Record). The MCP must transform these expressions into explicit I/O operators called I/O descriptors. An I/O descriptor allows the MCP to communicate directly with a peripheral device via the soft I/O routines of GISMO. GISMO manages the execution of these operators by the I/O subsystem. Each I/O descriptor provides such information as the type of I/O operation requested, source or destination memory addresses, the device which is to execute the operators, and space for result information used when control is passed back to the MCP. Certain other fields vary with the type of descriptor and contain information peculiar to its specific function.

Any number of I/O descriptors may be linked together to form a single "chain" and "dispatched" in one MCP operation to lessen the MCP's interaction with the I/O subsystem.

The transformation of logical I/O requests to physical I/O descriptor manipulation is discussed in the Demand Management section of this specification. The discussion below is intended to describe the operations performed upon the descriptor after it has been transformed. A programmatic description of an I/O descriptor is given below. This particular descriptor is typical of one which might be constructed for a disk file.

```
DEFINE IO.DESC.DECLARATION AS #%
        DECLARE 01  DUMMY                              REMAPS IO.DESC
              ,     02  IO.RESULT                       WORD
              ,       03  IO.COMPLETE                  BIT (1)
              ,       03  IO.EXCEPTION                 BIT (1)
              ,       03  IO.PACK.NOT.READY            BIT (1)
              ,       03  IO.DATA.ECC.ERROR            BIT (1)
              ,       03  FILLER                       BIT (1)
              ,       03  IO.MEM.PARITY.ERROR          BIT (1)
              ,       03  IO.WRITE.LOCKOUT             BIT (1)
              ,       03  FILLER                       BIT (2)
              ,       03  IO.ADDRESS.PARITY.ERROR                BIT (1)
              ,       03  IO.SECTOR.ADDRESS.ERROR                BIT (1)
              ,       03  IO.SEEK.TIMEOUT              BIT (1)
              ,       03  FILLER                       BIT (3)
              ,       03  IO.TRANSMISSION.PARITY.ERROR   BIT (1)
              ,       03  IO.RESULT.BIT.17             BIT (1)
              ,       03  IO.PORT.RS                   BIT (3)
              ,       03  IO.CHANNEL.RS                BIT (4)
              ,     02  IO.LINK                         ADDRESS
              ,     02  IO.OP                           WORD
              ,       03  IO.OP.OP                     BIT (3)
              ,       03  IO.OP.M                      BIT (1)
              ,       03  IO.OP.W                      BIT (1)
              ,       03  IO.OP.V                      BIT (1)
              ,       03  IO.OP.E                      BIT (1)
              ,       03  IO.OP.D                      BIT (1)
              ,       03  IO.OP.NNN                    BIT (3)
              ,       03  FILLER                       BIT (5)
              ,       03  IO.OP.P                      BIT (1)
              ,       03  FILLER                       BIT (3)
              ,       03  IO.OP.UNIT                   BIT (4)
              ,     02  IO.BEGIN                        ADDRESS
              ,     02  IO.END                          ADDRESS
              ,     02  IO.DISK.ADDRESS                 ADDRESS
              ,     02  IO.M.EVENTS                     BIT (8)
              ,       03  IO.M.EVENTS.IOC              BIT (1)
              ,       03  IO.M.EVENTS.SIOC             BIT (1)
              ,       03  FILLER                       BIT (1)
              ,       03  IO.M.EVENTS.INT.M            BIT (1)
```

```
,      03  IO.M.EVENTS.S.INT.SENT    BIT (1)
,      03  IO.M.EVENTS.M.INT.SENT    BIT (1)
,      03  FILLER                    BIT (1)
,      03  IO.M.EVENTS.INT.S         BIT (1)
,    02  IO.MCP.IO                   BIT (16)
,    02  IO.FIB                      ADDRESS
,    02  IO.FIB.LINK                 ADDRESS
,    02  IO.BACK.LINK                ADDRESS
,    02  IO.PORT.CHAN                BIT (7)
,      03  IO.PORT                   BIT (3)
,      03  IO.CHANNEL                BIT (4)
,    02  IO.BEEN.THRU.ERROR          BIT (1)#;
```

GISMO = THE I/O DRIVER

With the exception of the Multi-Line Control used on Data Communications configurations, on the B1000 hardware the I/O controls have no direct connection with main memory. All data transfers between the controls and memory must go through the processor. GISMO is a set of micro-coded routines whose primary function is to interface between the MCPs and the actual hardware. This allows the MCPs to view the I/O subsystem as an I/O processor. The MCP can initiate I/O Descriptors and GISMO will handle initiation of the control, data transfer and termination. The MCPs can queue several descriptors for execution by a control, by properly setting the link fields in the descriptors, and GISMO will initiate each one in turn.

User programs make requests to the Micro MCP, and sometimes the Micro MCP must ask that the request be handled by the S.MCP, but in either case, the MCP will pass the request to GISMO who in turn will pass it on to the I/O control.

The I/O subsystem allows fifteen controls or channels to be connected to any machine. After GISMO initiates a control, it does not wait for completion of the operation but returns control to its caller. Consequently, one, and possibly more operations may be in process on the machine at any given time. At any given moment, however, when GISMO is executing it may only address one control.

The primary communication between the MCPs and GISMO is through the I/O descriptors. The S.MCP will initiate I/O operations using the DISPATCH S-operator and the M.MCP contains micro-code to perform a similar function. This S-operator requires two parameters, the port and channel of the device being addressed and the memory address of the descriptor. The I/O decsriptor contains all of the information needed by GISMO for the operation.

An I/O descriptor is usually located by its "Reference Address", the memory address of the result descriptor field of the I/O descriptor. The result descriptor field is often referred to as the "RS field", or Result Status field. All of the descriptors associated with a given control will be linked together in memory, by setting IO.LINK to the memory address of the RS field of the next descriptor. The descriptors are also linked in the reverse direction, using the IO.BACK.LINK field, to facilitate adding and deleting descriptors. A link field may not be zero, but a descriptor may be linked to itself.

The Reference Address points to the RS field. Each RS field is twenty-four bits in length. The bits in the RS field have different meanings at different times. GISMO is most concerned with the setting of the bits when the I/O is initiated. The MCPs are more concerned with the setting of the bits when the I/O is complete. When the descriptor is ready for initiation, the RS field is formatted as shown in the following diagram. This field is usually referred to as the result status field when the descriptor is ready for execution or is in process and as a result descriptor field when the I/O operation is complete.

Bits 0-1 - RS Status Bits

00 - Ready to be Executed
01 - I/O Currently in Process
10 - I/O Complete with no Exception
11 - I/O Complete with Exception

Bits 2-11 - Gismo Toggles

MCPs may not alter any bits in this field if RS Status = 01.

Bits 12-14 - Port to which this I/O is directed. (Not used)

Bit 15 - Interrupt requested on I/O Completion.

Bit 16 - High-Priority interrupt requested on I/O Completion.

Bits 17-19 - Port to which interrupts are to be sent upon I/O Completion (Always Processor Zero).

Bits 20-23 - Channel on which I/O is to be performed.

The leftmost bit of an RS field is always set when the operation is complete. Consequently, storing a result descriptor locks the descriptor to GISMO. The MCP may lock a descriptor as well, if

the status field is not 01. Gismo will only initiate "ready" descriptors, those whose status bits are equal to 00. When the operation is initiated, GISMO sets the status bits to 01. The GISMO toggles area is used by GISMO when an I/O is in process to store information which it needs concerning the operation.

CHANNEL TABLE

Another structure associated with peripheral management is the channel table. There is one channel table for each port and each element of the table describes one channel of that port. While GISMO uses the I/O descriptor to communicate directly with the I/O subsystem, the channel table is a structure for passing information between the MCP and GISMO. The channel table reflects the status of a particular channel. Certain information is passed to GISMO during a "dispatch" operation and is used by soft I/O in managing the execution of that operation. Certain fields are updated before GISMO passes control back to the MCP which direct the course of action the MCP will take. A programmatic description is given below:

```
DEFINE CHANNEL.TABLE.DECLARATION AS # %
DECLARE 01 DUMMY REMAPS CHANNEL.TABLE %
,    02 CHANNEL.BUSY            BOOLEAN %
,    02 CHANNEL.PENDING         BOOLEAN %
,    02 CHANNEL.EXCEPTION       BOOLEAN %
,    02 CHANNEL.PAUSE           BOOLEAN %       0 = TAPE, DISK, CAS
,    02 CHANNEL.OVERRIDE        BOOLEAN %
,    02 CHANNEL.EXCHANGE        BOOLEAN %
,    02 CHANNEL.OLD.MODE        BOOLEAN %
,    02 CHANNEL.INTEGRITY       BOOLEAN %
,    02 CHANNEL.NO.HALT         BOOLEAN %
,    02 FILLER                  BIT (3) %
,    02 CHANNEL.TYPE            BIT (4) %            DEVICE TYPE FOR DUMP
 %                              TYPE = 0 = SERIAL DEVICE
 %                              TYPE = 1 = DISK
 %                              TYPE = 2 = TAPE
 %                              TYPE = 3 = CASSETTE
,    02 CHANNEL.LAST            BOOLEAN %          DELIMITS CHAN TABLE
,    02 CHANNEL.EXCHANGE.PC     BIT (7) %
,      03 CHANNEL.EXCHANGE.P    BIT (3) %
,      03 CHANNEL.EXCHANGE.C    BIT (4) %
,    02 CHANNEL.REF.ADDR        ADDRESS %
; # ; %
```

In the CHANNEL.TABLE, BUSY is set and reset by GISMO only. It is set when the control is busy. PENDING is also set and reset by GISMO. It is used on tape and disk devices only and it tells GISMO to continue linking through the head of the queue. EXCEPTION is used on all devices except tape and disk. It causes

GISMO to inhibit dispatch operations on the channel until a prior exception condition has been handled by the MCP.

PAUSE is also known as the TIMER bit. It is set by the MCP and it never changes. It causes GISMO to issue a dispatch to the channel at each 100 millisecond timer interval and is used to implement TEST.AND.WAIT operations on tape and disk controls. This is discussed in more detail later.

The OVERRIDE bit is used on all devices and causes GISMO to reset BUSY, PENDING and EXCEPTION when a new operation is dispatched. It is set by the MCPs and reset by GISMO. Essentially, it causes GISMO to override an existing operation with a new operation.

The EXCHANGE bit is set by the MCP and it never changes. It is used on tape and disk controls only and it means that the information in EXCHANGE.PC is valid, that there is another control connected to this control by a hardware exchange. The OLD.MODE bit, also known as the PAUSE bit, is also set by the MCP and never changes. It is set for Single-Line Controls and for Disk Cartridge Control One. It causes GISMO to pause for 100 milliseconds when a locked descriptor or a Pause I/O descriptor is encountered. If this bit is not set, GISMO will stop in this circumstance on these controls.

The INTEGRITY bit is set by the MCP when the channel table entry is initialized. It is also used by the MCP to stop GISMO from linkimg on the channel.

The TYPE field is used only by the Dump Analyzer program. It is necessary because the analyzer may have no other means of determining this information. The REF.ADDR field contains the address of the descriptor that is in process on this channel. It is considered the head of the queue by GISMO.

GISMO/HARDWARE INTERFACE

The I/O descriptor contains most of the information GISMO needs to accomplish an I/O operation. In the actual hardware interface, the OP, BEGIN, END, DISK.ADDRESS and ACTUAL.END fields are used. The ACTUAL.END field is twenty-four bits in length and immediately preceeds the RS field in each descriptor. It is not shown in the preceding I/O descriptor diagram. The field is used by GISMO while the operation is in process to store the memory address of the data that is to be transferred to or from the memory buffer. When the operation is complete, ACTUAL.END will contain the address of the next bit that data would have been

transferred to or from.

Each control is able to buffer, or store, a certain amount of data to be transferred. The amount varies among the devices. For some devices, such as the card reader and line printer, it is a full record. For others, the size of the buffer may vary and each contol may contain a portion of the data. Disk controls, for example, are equipped with a certain number of 180-byte hardware buffers. The amount of data that may be contained in the controls and the procedures that GISMO must follow in the execution of an operation are fixed when the control is designed and do not change afterward.

CA/RC CYCLES

The hardware in the processor that is used by GISMO is the Command Register, the Data Register and the Service Request level. The Command Register is used to send information to a control, the Data Register to receive from the control and the Service Request level indicates that a control needs attention from GISMO.

Most transactions with the control consist of a Command-Activate/Response-Complete (CA/RC) cycle. Data or command information is sent out to a control with a CA. Control information or data is returned with a RC.

PROCESSOR I/O INSTRUCTIONS

The processor instructions whch GISMO uses to accomplish an operation are:

TEST STATUS

GISMO requests and the control returns its current status count and the device ID. GISMO uses this information to decide what to do next.

TEST & CLEAR

This operation clears the control.

TEST SERVICE REQUEST

GISMO requests, and the processor returns, a mask of all channels that are currently requesting service.

TERMINATE DATA

This operator is used to terminate data transfer when the media, disk and tape for example, has no fixed record size.

TRANSFER OUT A

Moves one or two bytes of data from memory to the control for output to the device. Data is sent at CA time; the control returns its status at RC time.

TRANSFER OUT B

Moves three bytes of data from memory to the control for output to the device.

TRANSFER IN

Moves one, two or three bytes of data from the control to main memory on input operations. The data is sent at RC time. When one or two bytes is transferred, the control also sends its status.

SERVICE REQUEST

The Service Request level is a toggle in the processor which is settable by any control. It is OR-ed into the "Any Interrupt" toggle. Each Interpreter, prior to executing an S-operator, will test the Any Interrupt toggle and, if it is set, transfer control to GISMO instead. GISMO will determine what caused the toggle to be set. In this case, it will discover that Service Request is raised.

It will then do a TEST SERVICE REQUEST CA/RC cycle. The RC will return a mask of all controls that are currently requesting service. GISMO will select the highest channel from this mask and begin handling that control. Conrols are usually in status count 11 or 18 when they raise Service Request. This status indicates that the control is ready to send a Reference Address to GISMO. GISMO accepts the Reference Address and uses it to locate I/O descriptor in memory.

GISMO will then do a TEST STATUS CA/RC cycle to detemine what service the control is requesting. Once the requested service has been performed, and the control no longer is requesting service, GISMO will again perform a TEST SERVICE REQUEST CA/RC cycle. It will continue handling Service Requests from various controls until the TEST SERVICE REQUEST returns all zeros. GISMO then returns control to the Interpreter that was interrupted.

STATUS COUNTS

The Status Count returned by a control is the primary means in which GISMO determines what is to be done next in an I/O operation. Operations may consist of sending the Op code and file address, sending the Reference Address, receiving the Reference Address, sending or receiving data and receiving the result. Various controls perform these steps in different orders.

All controls begin in Status Count 1 and return to Status Count 1 after Status Count 23. Each Status value has a particular meaning. Some counts always appear in series together. All controls begin an operation by going through Status Counts 1 through 6. A simplified table of the allowable Status Count transitions is shown in the table below.

To send each of the twenty-four bit fields OP, DISK.ADDRESS and Reference Address, three TRANSFER OUT operations are used, each CA/RC sending one byte. For each TRANSFER OUT, the Status Counter advances by one. Similarly, to receive either the Result Descriptor or the Reference Address, three TRANSFER IN operations are used, each CA/RC receiving one byte.

Status Count	Meaning
0	Control Not Present
1	Cleared (Initial) State
1, 2, 3	Ready to Receive OP, Bytes 1, 2 and 3
4, 5, 6	Ready to Receive DISK ADDRESS, Bytes 1, 2, 3
7, 8, 9	Ready to Receive Reference Address, Bytes 1, 2, 3
10	Busy (Operation in process). From 10, Controls usually go to Status 11 or 18 and raise Service Request.
11, 12, 13	Ready to Send Reference Address, Bytes 1, 2, 3.
14	Ready to Receive Data (output)
15	Ready to Send Data (input)
16	End of Hardware Buffer - Ready to Send or Receive Last Byte. More Buffers Remain.
17	End of Hardware Buffer and Last Buffer.
18, 19, 20	Ready to Send Reference Address, Bytes 1, 2, 3. Implies that a Result Descriptor is to Follow.
21, 22, 23	Ready to Send Result Descriptor, Bytes 1, 2, 3.

Table X.X - Typical Control Status Counts and their Meaning

DATA TRANSFERS

GISMO transfers data to and from the control in one or more iterations; each iteration will involve only one control buffer. For some devices, there is only one buffer and this buffer will always contain the full physical record. GISMO will only perform data transfer once per I/O operation for these controls. Other controls have physical records of undefined length; for these controls, there are usually multiple buffers of fixed length in the control and each iteration of GISMO will fill or empty one of these buffers.

Whenever Service Request is raised and GISMO is invoked, the requesting control will first send the reference address. GISMO will then test the control's status. If the control is in Status 14 or 15, GISMO will begin data transfer. For each operation, data transfer will continue until either the control's buffer is empty or the END address of the I/O descriptor is reached. In the first case, the control will have gone to Status 7 after the last data character(s). GISMO will test its status, see that it is in Status 7 and send it the Reference Address, thus completing the iteration. In the latter case, on most controls, GISMO will send it a TERMINATE command. Some controls require data transfer to continue until the end of the control's buffer. On input, GISMO will accept the remaining data from these controls but will not store it in memory. On output GISMO will send blanks to these controls.

Data is always transferred to a control in one, two or three byte portions. Most "Serial" devices, such as printers and card devices, use one byte transfers. This data transfer is performed from a loop within GISMO which consists of a CA/RC cycle, transferring one data byte, until the control's buffer is full or the END address is reached. A buffer full condition is detected by the control sending or receiving the last data byte in Status Count 16 or 17.

Many disk and tape controls transfer data two bytes per CA/RC. Disk input and output is always terminated by GISMO when the END address is reached, possibly in the last of multiple disk sectors. When the record length is an odd number, GISMO will normalize the last byte as required. On output operations, the control will pad the remainder of the last buffer (and hence sector) with zeros.

Tape output, possibly in the last of multiple buffers, is also terminated by GISMO when the END address is reached. When the

physical record size is an odd number of characters, GISMO will normalize the last byte for the last CA/RC cycle. It will send a TERMINATE command, followed by a special command which will indicate "odd character count" to tell the control that the last data transfer consisted of one byte only. Tape input operations will terminate either when the END address is reached or when the end of the physical record is encountered, which may be in the last of multiple buffers. If the end of the physical record occurs and the length of the record is an odd number of characters, the control will set a flag in the RC portion of the last CA/RC cycle. GISMO will then normalize the last byte of the record.

All disk pack controls, the 5N head-per-track control and all phase-encoded tape controls use three byte data transfers. In this case only, an exception is made to the general rule that all transactions involve one CA and one RC. On these controls, one CA may be followed by one or more RCs. This is accomplished as follows.

Prior to entering the transfer loop, on input, GISMO will use a special CA/RC cycle to ask the control how many butes it has to send. It will then initiate the transfer loop with a CA and continue it with as many RCs as are required, receiving twenty-four bits of data on each RC. For output, GISMO will tell the control how many bytes it has to send. It will then initiate the transfer loop with a CA command of TRANSFER OUT B, and continue it with as many RCs as are required, sending the data out with the RC.

I/O CHAINING

The I/O subsystem of the B1000 system does not use queues for I/O operations. Using the facilities presented in the preceding, it connects all I/O descriptors that are directed to the same control, or group of controls connected by an exchange, in a circular chain. This eliminates the necessity of an I/O complete interrupt being directed to the MCP, provided the producer of I/O requests, most often a user program, does not produce the requests faster than they can be satisfied. In other words, if the I/O subsystem is completing operations before they are actually required by the user, then the user will never need to wait on the completion of an I/O request and the MCP will never have to suspend the program waiting for such a completion.

Even if this isn't the case, if the user program is forced to wait upon the completion of his I/O requests, the amount of processing that must be done to accomplish the suspension and to reinstate the program upon completion is minimized using

chaining. The processing is limited to only that which is concerned with program execution and no processing is required to tell the I/O subsystem what it should do next. This information is already contained in the I/O descriptor.

For all devices except tape and disk, then, the MCP constructs a circular chain of descriptors in memory. GISMO executes the requested operations in turn, as each descriptor is unlocked by the MCP. Upon encountering a locked descriptor, GISMO simply pauses or stops until the descriptor is unlocked. This will occur when the user program next executes an I/O request or when the file is closed for any reason. If the program must wait upon an operation, an I/O complete interrupt is requested, using the appropriate bit in the RS field, and the program is suspended pending the occurrence of the interrupt.

DISK I/O CHAINING

The disk I/O subsystem operates somewhat differently from the operation just described. Since each disk I/O descriptor contains a disk address field, it is not necesary for the operations to execute in any particular order. Various means are provided in the software to prevent any contention problems that might arise. It may be noted that these same means are necessary on I/O subsystems which utilize queueing instead of chaining.

All I/O descriptors for all disk controls that are connected to the system are connected in the same chain. If the system is equipped with more than one control, then each Channel Table entry will point to the head of the chain. If GISMO encounters a descriptor which is not ready for execution or which is already in process, specified by the first two bits of the RS field being set to anything other than 00, it does not stop or pause but continues to the next descriptor in the chain. Also, if an exception condition occurs, GISMO does not stop or pause as it does on other controls. Both of these actions are specified by the CHANNEL.NO.HALT bit in the Channel Table.

Since GISMO continues linking in both of the cases mentioned above, it must know when it has examined all of the descriptors in the chain. When it has examined all of the descriptors, it must stop to free the processor for other execution. To accomplish this, the REF.ADDR field in the Channel Table is used to mark the beginning of the chain. When a disk operation is dispatched by the MCP, the reference address passed by the dispatch is discarded and the REF.ADDR field is used instead.

In order to operate properly with dispatch operations occurring in an order different from the order of the descriptor link fields, GISMO must be able to override stopping when it has been through the entire chain once. For example, if descriptors A, B, and C are present in the chain and if B is dispatched, GISMO will link to and initiate B. If, during the time that B is in process, A is dispatched, GISMO must link past C and the REF.ADDR field and find and initiate A.

To accomplish this, the PENDING bit in the Channel Table is used. This bit is set by a dispatch operation and reset by GISMO. If GISMO arrives at the descriptor addressed by the REF.ADDR field and if the PENDING bit is set, it does not stop but resets PENDING and continues linking. If PENDING is already reset at this point, then GISMO stops.

Since all descriptors for all disk controls are maintained in the same chain, GISMO must be able to recognize descriptors which are addressed to controls different from the one it is handling. This is accomplished using the IO.CHANNEL.RS field of the I/O descriptor. Upon encountering an unlocked I/O descriptor, GISMO compares this field to the channel it is executing upon and if the two are not equal, it does not mark the descriptor in process but continues linking.

DISK I/O OVERLAPPED SEEKS

When an I/O operation is initiated on a moveable arm disk device and the arm is presently positioned to a cylinder different from the one specified in the descriptor, it is necessary to reposition the arm to the proper cylinder. This operation is known as a "seek". On the B1000 system, all seek operations are implicit; there is no explicit Seek operation in the hardware. The MCPs initiate disk I/O operations without regard for the current arm position and, if arm movement is required, it is accomplished by GISMO, the control and the device without the MCP's participation. The MCP does not know that a seek is being performed or required.

On this system, all seek operations are "overlapped". This means that the arm of any given drive may be in motion simultaneously with the arm of any other drive(s). Also, the control may be performing data transfer or any other operation while the arms are in motion.

This is accomplished by the control returning a result descriptor with Bit 17, IO.RESULT.BIT.17, set to zero. Esssentially, this informs GISMO that some special action is necessary and that

GISMO should not store the result descriptor in memory. In this particular case, the control also informs GISMO that the selected drive is now seeking. GISMO will initiate no further operations upon that drive until informed, by the hardware, that the seek operation has completed.

DCC-2 (Cartridge) and all disk pack controls notify GISMO that a seek operation has completed by raising Service Request while in Status Count 1. GISMO will again send the descriptor to the control and this time, after any required latency period, data transfer will occur. DCC-1 does not notify GISMO when a seek operation has completed but must be "polled" periodically by GISMO. The pause time period for DCC-1, the time between the poll operations, is two milliseconds.

The Disk Subsystem Controller (DSC) offerred on GEM processors introduces some exceptions to the statements above. These exceptions will be defined in a subsequent version of the specification.

TAPE I/O CHAINING

The chaining of I/O descriptors for magnetic tape controls is perhaps the most complex of the three basic types. The complexity is caused by the fact that tape I/O descriptors directed to each separate tape unit must be executed in logical sequence and there may be several such units attached to the same control(s). It doesn't matter which unit GISMO addresses next but the descriptor that is used to address the unit must be the next logical descriptor in the "subchain" for that unit. It is therefore necessary to break the channel chain into subchains, with one subchain for each physical unit, and to implement a means of remembering the next logical descriptor that must be used within each subchain.

Both of these requirements are satisfied by the Lock descriptor. Lock is a pseudo I/O operation which is handled completely by GISMO and actually causes no physical I/O operations. It also serves as a means of resolving contention problems between the MCPs and GISMO and between two or more tape controls which are attached to the same units by an exchange. Lock operates as described below.

The MCP, when the system is Clear/Started, constructs a tape chain with one Lock descriptor for each unit connected to the system. The ACTUAL.END field of a Lock descriptor is not used and the LINK field will contain the memory address of the next Lock descriptor. The BEGIN and END address fields of the Lock

descriptor will contain the address of the TEST.AND.WAIT I/O descriptor that the MCP uses to monitor the status of each unit. This is discussed in a later paragraph.

When a file is opened on a tape unit, the MCP changes the BEGIN and END address fields in the Lock descriptor. The MCP now constructs a subchain for the unit which will consist of one I/O descriptor for each buffer requested by the user. The BEGIN and END addresses of the Lock descriptor will be set to the memory address of the first physical I/O descriptor in the subchain and the TEST.AND.WAIT descriptor will be removed from the subchain. The BEGIN address field will not be altered from this point until the file is Closed. The END address will be modified by GISMO each time it executes an operation in the subchain. In effect, The END address field is used to remember the next logical operation that is to be performed on the unit.

The LINK fields in each I/O descriptor in the subchain will all address the next physical descriptor in the subchain, as they do for all other controls. An exception to this is the last physical descriptor in the subchain. The LINK field of this descriptor will contain the address of the Lock descriptor for that unit. This prevent one unit from monopolizing the entire control; it insures that GISMO will periodically determine if there is anything to be done on the other units.

The REF.ADDR field of the Channel Table entry for a tape chain will contain the address of the first Lock descriptor in the chain. Gismo, upon receiving a Dispatch for a tape control, will discard the Reference Address passsed and start at the address provided by the REF.ADDR field. GISMO first attempts to lock the Lock descriptor by swapping 01 into the first two bits of the RS field. If successful, it fetches the address in the END field of the Lock descriptor and proceeds to that address. If this descriptor is unlocked, it begins the operation specified. If not, it returns to the Lock descriptor and stores the address, which it previously fetched from the END address field back into the END address field.

Assume now that the descriptor at the address fetched from the END field of the Lock descriptor was unlocked. GISMO begins this operation and, assuming that the operation cannot be completed without some intermediate Service Requests, returns to the Lock descriptor and continues linking through the chain. Eventually, the control will raise Service Request and reference the initiated descriptor. Upon completion of that descriptor, GISMO will store a result and fetch the LINK field of the descriptor. It will then proceed to the new descriptor and again check to see if it is locked. If it is, GISMO returns to the Lock descriptor for the unit and stores the new address in the END address field.

The new descriptor now becomes the next logical descriptor to be executed on that unit. In this manner, GISMO effectively maintains a logical sequence of operations that are to be performed on any tape unit.

It may be noticed from the foregoing that there is no possibility of conflict for a unit between two or more controls connected by an exchange, since GISMO first attempts to lock the Lock descriptor before proceeding down a subchain. Similarly, the MCP must lock the subchain before altering any descriptor in the subchain.

MONITORING OF PERIPHERAL STATUS

The MCP attempts to monitor the status of all peripheral devices that are attached to the system. To do this, it must remember the status of each device and maintain a certain amount of information about each. The major portion of the information about all of the devices connected is maintained in the I/O Assignment Table (IOAT).

I/O ASSIGNMENT TABLE

The I/O Assignment Table (IOAT) allows the MCP to keep track of all peripheral units except the system's SPO and those devices associated with data communication. Each unit is identified by port, channel, and unit numbers as well as by a symbolic name. Various fields reflect the status of the unit (e.g., AVAILABLE, SAVED, REWINDING, LOCKED). A programmatic description is given below:

```
DEFINE IOAT.SIZE AS #512#;
DEFINE IOAT.DECLARATION      AS          #  %G L O B A L    I O A T
DECLARE 1 DUMMY REMAPS IOAT.
    02   UNIT.INITIAL                    BIT (66), %
         03   UNIT.HDWR                  BIT (6),
         03   UNIT.PCD                   BIT (12), %
           04 UNIT.PORT.CHANNEL          BIT (7), %
             05 UNIT.PORT                BIT (3), %
             05 UNIT.CHANNEL             BIT (4), %
           04 FILLER                     BOOLEAN, %
           04 UNIT.UNIT                  BIT (4), %
         03   UNIT.NAME                  CHAR (6),
    02   UNIT.LABEL.ADDRESS              DSK.ADR,
         03   FILLER                     BIT (12),
         03   UNIT.PACK.INFO             ADDRESS,
    02   UNIT.RS                         ADDRESS,% USER LIMIT REGISTER
    02   UNIT.FLAGS                      BIT(36),
```

```
      03    UNIT.AVAILABLE                BOOLEAN,
      03    UNIT.AVAILABLE.INPUT          BOOLEAN,
      03    UNIT.AVAILABLE.OUTPUT         BOOLEAN,
      03    UNIT.WAIT.FOR.NOT.READY       BOOLEAN,
      03    UNIT.TEST.AND.WAIT            BOOLEAN,
      03    UNIT.SAVED                    BOOLEAN,
      03    UNIT.REWINDING                BOOLEAN,
      03    UNIT.EOF.SENSED               BOOLEAN,
      03    UNIT.LOCKED                   BOOLEAN,
      03    UNIT.LABEL.SENSED             BOOLEAN,
      03    UNIT.PRINT.BACKUP             BOOLEAN,
      03    UNIT.PURGE                    BOOLEAN,
      03    UNIT.LOCK.AT.TERM             BOOLEAN,
      03    UNIT.TO.BE.SAVED              BOOLEAN,
      03    UNIT.FLUSH                    BOOLEAN,%  FLUSH TO EOF
      03    UNIT.TAPEF                    BOOLEAN,
      03    UNIT.DISKF      BOOLEAN,
      03    UNIT.STOPPED    BOOLEAN,
      03    UNIT.TRANSLATE                BOOLEAN,
      03    UNIT.CTRL.CARD.USING          BOOLEAN, %
      03    UNIT.REMOTE.JOB               BOOLEAN,
      03    UNIT.CLOSED                   BOOLEAN, %
      03    UNIT.CLEARED                  BOOLEAN,
      03  UNIT.MULTI.FILE                BOOLEAN, %
      03  UNIT.EOT                       BOOLEAN,
      03    UNIT.TAPE.FILE.STATUS         BIT(3),% 0 = NOT RELEVANT(_ANSI)
                                                  % 1 = BOV(BEG OF VOLUME
                                                  % 2 = BOF(BEG OF FILE)
                                                  % 3 = EOV(END OF VOLUME)
                                                  % 4 = EOF(END OF FILE)
                                                  % 5 = PFB(PROCESS FILE BLK
                                                  % 7 = UNDEFINED
      03 UNIT.TAPE.XCH                  BOOLEAN,% FOR MIS-MATCHED UNITS
      03 UNIT.NO.TRANS.TBLE BOOLEAN,%PC-5
03    UNIT.OFFLINE.YET.IN.USE           BOOLEAN,%FOR ASSIGNED UNITS.
      03    UNIT.AUDIT                  BOOLEAN,  % DMS AUDIT TAPE
      03    UNIT.RESERVED.BY.AB         BOOLEAN,%  AUTO BACKUP 6.1
      03 UNIT.LABEL.OP                  BIT(3),% 0=@00E00X@ ODD TRANS
                                                  % 1=@00C00X@ ODD NO TRANS
                                                  % 2=@00600X@ EVEN TRANS
                                                  % 3=@00400X@ EVEN NO TRANS
 02   UNIT.DRIVE.TYPE                   BIT(4),% DISK ONLY
          %  VALUE      DCC1/2/3   DPC1/2   DFC1      DFC3
          %     0         32X203      N/A      N/A       N/A
          %     1         32X406      215    SYS.MEM    5N
          %     2         64X203      225      N/A       N/A
          %     3         64X406      N/A     1C-3       N/A
          %     4           N/A       207     1C-4       N/A
          %     5           N/A       205     1A-3       N/A
          %     6           N/A       206     1A-4       N/A
          %     7           N/A       N/A      N/A       N/A
 02   UNIT.STATUS                        BIT (15),
 02   UNIT.TO.BE.POWERED.OFF             BOOLEAN,
 02   FILLER                             BIT(7),
```

```
    02   UNIT.JOB.NUMBER                    BIT(16),
    02   UNIT.FIB.ADDRESS                   ADDRESS,
    02   UNIT.LABEL.TYPE                    BIT (2),
                  %     0 = OMITTED
                  %     1 = BURROUGHS
                  %     2 = USASI
                  %     3 = INSTALLATION
    02   UNIT.TRANS.TBLE.ID                 BIT(8),  %PC-5 TRAIN ID
    02   FILLER                             WORD,% PLEASE DO NOT DISTURB
    02   UNIT.TEST.DESC                     BIT (DESCRIPTOR.SIZE);
#;    %   D E L I M I T    I O A T    D E F I N E
```

The entire IOAT is constructed by the MCP when the system is Clear/Started. During the Clear/Start operation, the MCP directs a Test descriptor to each of the controls that are connected to the system. When it discovers a control that may have more than one unit connected to it, it sends a Test descriptor to each possible unit and makes one entry in the IOAT for each unit that is connected.

The UNIT.HDWR field in the IOAT will contain the hardware identifier returned by the test descriptor. The following is a list of hardware types and pseudo-types that are supported by the MCP. Pseudo-types are used in the device assignment process to indicate generic types, such as "any magnetic tape device" which would include seven-track, nine-track, phase encoded, NRZ and so forth.

DEVICE	FILE STMT	HDWR TYPE
Reserved		00
80 col READER.PUNCH.PRINTER	DATA.RECORDER.80	01
80 col CARD PUNCH	CARD.PUNCH	02
Reserved		03
FDC.1		04
96 col READER PUNCH PRINTER	READER.PUNCH.PRINTER	05
PAPER TAPE READER	PAPER.TAPE.READER	06
PAPER TAPE READER-1	PAPER.TAPE.READER	07
PRINTER	PRINTER	08
READER SORTER-2	READER.SORTER.2	09
READER SORTER	READER.SORTER	10
DISK FILE (Any head per track)	DISK.FILE	11
DFC-1	DISK.FILE.1	12
DCC-2	DISK.CARTRIDGE	13
DCC-1	DISK.CARTRIDGE	14
DPC-1	DISK.PACK.10	15
DISK PACK (DCC-1, DCC-2, DPC-1)	DISK.PACK	16
DISK (Any disk)	DISK	17
DFC-3 (5-N)	DISK.FILE.3	18
96 col READER	READER.96	19
PAPER TAPE PUNCH	PAPER.TAPE.PUNCH	20
80 col CARD READER	CARD.READER	21
SPO-1		22
SPO-2	CRT SPO	23
TAPE 9 TRK NRZ	TAPE.9	24
TAPE 7 TRK NRZ	TAPE.7	25
TAPE PE (9 TRK)	TAPE.PE	26
TAPE (Any tape)	TAPE	27
TAPE.9 (Any 9 TRK tape)	TAPE.9	28
Reserved		29
CASSETTE	CASSETTE	30
LPC-5	PC.5	31
QUEUE FILE	QUEUE	62
REMOTE FILE	REMOTE	63

Table 3.x - Hardware types supported by MCP

In the table above, the File Statement column (FILE STMT) is for use in the MCP's FILE Control Card and is explained in the Software Operational Guide. Generic hardware type numbers are not stored in the IOAT. Rather, the actual identifiers returned by the hardware are used.

UNIT MNEMONICS

Unit mnemonics are also assigned by the MCP during the Clear/Start process. These mnemonics allow the operator and the MCP to identify devices uniquely. The table below lists the form of the mnemonic that will be assigned to the various types of devices.

Card Reader	CRx
Card Punch	CPx
Data Recorders	CDx
Printers	LPx
Tape Units	MTx
Disk (head-per-track)	none
Disk Pack	DPx
Disk Cartridge	DCx
Paper Tape Readers	PRx
Paper Tape Punches	PPx
Reader-Sorters	RSx
Cassettes	CSx
Flexi-Disk	FDx

All units will be assigned a three-character mnemonic which begins with the first two letters listed in the table above. The third character will be unique to the unit. The first unit of that type encountered by the MCP during the Clear/Start operation is assigned the letter "A", the second "B" and so forth. Assignment proceeds alphabetically and the mnemonic assigned does not change unless the system configuration changes.

The assigned unit mnemonic is stored in the IOAT in the UNIT.NAME field. The entire IOAT is maintained in memory. To minimize storage requirements, some information which relates to the unit is not stored in the IOAT but is maintained on disk. File Identifiers and any other information which is seldom used by the MCP are stored in an INTERNAL.LABEL field on disk. The disk address of this field is maintained in the IOAT in the UNIT.LABEL.ADDRESS field. Information in this field is typically updated by the STATUS procedure in the MCP.

The STATUS procedure is executed whenever the Ready status of an unassigned device changes. The MCP is made aware of a status change by TEST.AND.WAIT I/O operators. These operators do not

truly wait on a unit status change but this function is emulated by GISMO.

TEST.AND.WAIT I/O OPERATORS

The MCP must know when a unit goes from a Not Ready condition to a Ready condition so that it can read the label on the media and update the INTERNAL.LABEL information on disk. It must know when a unit changes from Ready to Not Ready so that it can mark the unit unavailable and initiate a TEST.AND.WAIT.FOR.READY on the unit. TEST.AND.WAIT operations allow the specification of certain conditions for completion, such as Test and Wait for Ready, Not Ready, Ready to Transmit, Ready to Receive and so forth. GISMO will not consider the operation complete unless the specified conditions are met.

On disk and tape controls, which allow more than one unit per control, we cannot tie up the entire control with a Test and Wait operation to one unit. For DCC-2, all disk pack and all tape controls, the PAUSE bit in the Channel Table is used to implement a periodic test of all such units. At each 100 millisecond timer interval, GISMO searches through the Channel Table looking for entries with this bit set to zero. When such an entry is found, GISMO initiates that chain at the address specified by REF.ADDR, also in the Channel Table. During this execution, GISMO will initiate all Test operations encountered in the chain. If the conditions for completion specified in the operator have been met, GISMO will store the result descriptor returned by the operation and queue an interrupt for the MCP; the MCP always requests an interrupt in Test and Wait descriptors.

The MCP also sets the type field of this I/O descriptor, IO.MCP.IO, to a value which indicates "Status Change". In the MCP's I/O Complete procedure, which is invoked only when an interrupt is returned from an I/O operation, the value stored in IO.MCP.IO will cause invocation of the MCP's STATUS Procedure.

STATUS PROCEDURE

As mentioned previously, the STATUS Procedure is executed only when the status of an unassigned peripheral changes. If a peripheral is being used by a program and if it goes to a Not Ready condition, the situation is handled by the I/O Error Procedure. When an assigned peripheral goes from Not Ready to Ready, no action is required by the MCP since the Test and Wait descriptor executed in this case will have a LINK field set to the next logical operation to be performed on the device.

Peripheral devices which are capable of input operations usually have labels written on the media. The MCP is equipped to recognize several different label formats on disk and tape devices and it expects to read control instructions from all card devices which have input capabilities. Control instructions are discussed in the Software Operational Guide and in Product Specification 2219 0144, MCP Control Syntax and will not be discussed here. Essentially, when a card device becomes Ready for input purposes, the Status Procedure reads the first card and control is passed to the Control Card Procedure.

On disk and tape devices, when a unit becomes Ready, the Status Procedure attempts to read a label from the media. The following is a description of the various label formats, on disk and tape devices, the MCP is capable of recognizing.

DISK IDENTIFICATION - PACK LABELS

Every disk pack, disk cartridge, or head-per-track sub-system is identified by a standard "ANSI" pack label. This pack label, written in EBCDIC (8 bit code), is two pack sectors long (360 bytes), and occupies the first two sectors on a pack, I.E., cylinder 0, track 0, sectors 0 and 1. Sector 0 contains pack identification information and sector 1 is reserved for future implementation of pack security procedures. A programmatic description is given below:

```
DEFINE  PACK.LABEL.DECLARATION  AS #%
DECLARE 01 DUMMY REMAPS PACK.LABEL%
,    02 PL.VOL1                  CHAR (4)  %          "VOL1"
,    02 PL.SERIAL.NO             CHAR (6)  %          SERIAL (CAN) NUMBER
,    02 PL.ACCESS.CODE           CHAR (1)  %          ACCESS CODE
,    02 PL.ID                    CHAR (17) %          PACK ID
,         03 PL.NAME             CHAR (10) %
,         03 FILLER              CHAR (7)  %
,    02 PL.SYSTEM.INTERCHANGE    CHAR (2)  %          SYSTEM INTERCHANGE/CODE
                                           %  00 = INTERCHANGE
                                           %  17 = B1000 INTERNAL
                                           %  35 = B3500 INTERNAL
                                           %  ETC, ETC, ETC
,    02 PL.CODE                  CHAR (1)  %          PACK CODE 00 = SCRATCH
,    02 FILLER                   CHAR (6)  %
,    02 PL.OWNER.ID              CHAR (14) %
,    02 PL.TYPE                  CHAR (1)  %          "R" = RESTRICTED PACK
                                           %          "U" = USER PACK
                                           %          "S" = SYSTEM.PACK
,    02 PL.CONTINUE              CHAR (1)  %          CONTINUATION FLAG "C"
,    02 FILLER                   CHAR (26) %
,    02 PL.INT                   CHAR (1)  %
,    02 PL.VOL2                  CHAR (4)  %          "VOL2"
```

```
,    02 PL.DATE.INITIALIZED    CHAR (5)  %
,    02 PL.INIT.SYSTEM         CHAR (6)  %          INITIALIZING SYSTEM
,    02 PL.DISK.DIRECTORY      CHAR (8)  %          DIRECTORY ADDRESS
,    02 PL.MASTER.AVAIL        CHAR (8)  %          MASTER AVAILABLE TABLE
,    02 PL.DISK.AVAILABLE      CHAR (8)  %          WORKING AVAILABLE TABLE
,    02 PL.INTEGRITY           CHAR (1)  %          0 = NORMAL
                                         %          1 = RECOVERY REQUIRED
,    02 PL.ERROR.COUNT         CHAR (6)  %
,    02 PL.SECTORS.XD          CHAR (6)  %          REMOVED SECTORS
,    02 PL.TEMP.TABLE          CHAR (8)  %          TEMP TABLE LINK
,    02 PL.PCD                 CHAR (3)  % LAST PORT, CHAN, DRIVE
,    02 PL.ASSIGNED.TO.BPS     CHAR (6)  %    BASE PACK SERIAL NUMBER
```

In the case of disk devices, additional information, beyond that which can be stored in the IOAT, is required by the MCP for proper operation. The STATUS Procedure and others maintain this information in a reserved area in memory known as the Pack Information Table (PACK.INFO).

PACK INFORMATION TABLE

The pack information table is an MCP maintained linked list of all user disk packs and cartridges currently on line. It contains such information as the name, serial number, hardware unit, number of users, and addresses of the disk directory, available table, and temporary table. This structure allows a pack or cartridge to be externally referenced by name. A programmatic description is given below:

```
DEFINE PACK.INFO.DECLARATION  AS  #%
DECLARE 01 DUMMY REMAPS PACK.INFO,
     02 P.NAME                  NAME,
     02 P.SERIAL.NO             WORD,
     02 P.DISK.DIRECTORY        DSK.ADR,
     02 P.DISK.AVAILABLE        DSK.ADR,
     02 P.TEMP.TABLE            DSK.ADR,
     02 P.UNIT.NAME             CHAR (6),
     02 P.PCD                   BIT (12),
          03 P.PORT.CHAN        BIT (7),
          03 FILLER             BIT (1),
          03 P.DRIVE.NO         BIT (4),
     02 P.NO.USERS              BIT (8),
     02 P.NO.MPF.USERS          BIT (8),
     02 P.TO.BE.POWERED.DOWN    BOOLEAN,
     02 P.RESTRICTED            BIT (3),  % 0 = SYSTEM RESOURCE PACK
                                          % 1 = RESTRICTED
                                          % 2 = UNRESTRICTED USER
                                          % 3 = INTERCHANGE
     02 P.CONTINUE              BOOLEAN,  % 1 = CONTINUATION PACK
     02 P.SCRATCH               BOOLEAN,  % 1 = SCRATCH PACK
```

```
    02 P.FULL                    BOOLEAN,  % 1 = NO MORE AVL DISK
    02 P.XC                           BOOLEAN, %PACK HAS UNDERGONE XC.
    02 P.ASSIGNED.TO.BPS         WORD,     % ASSIGNED TO BASE PACK #
    02 P.BACK.LINK               ADDRESS,
    02 P.LINK                    ADDRESS;
#;%
```

TAPE LABELLING, INITIALIZATION AND PURGING

MCP II includes the capability to create and recognize two different forms of magnetic tape labels. The standard label format for the B1000 system will conform to that specified in the publication entitled "The American National Standard Magnetic Tape Labels for Information Exchange" which is dated 1969 and published by the American National Standards Institute, Inc. (ANSII). These labels are commonly known as "ANSII, Version 1" labels. It should be noted that "standard label format" for the system means that any program which requests standard labels in its file declaration will cause ANSII labels to be written when the file is assigned to magnetic tape, and the file is opened output. Users are allowed to create the label in ASCII if they so desire.

ANSII labels as implemented on the B1000 system contain several deviations from the standard as presented by the ANSII documents. The deviations are necessary in order to insure that we are compatible with the B6700 system. The most noteworthy deviation is the recording mode of the label itself; it is written in EBCDIC character code unless ASCII is specifically requested via the "SN" command.

ANSII label format, as implemented, consists of three physical blocks on the tape, followed by a tape mark. The first of the three blocks is known as the Volume Header. A programmatic description is presented below.

```
01 VOLUME.HEADER
   02 FILLER                     CHARACTER(4)
        %This field will always contain "VOL1"
   02 VOLUME.ID                  CHARACTER(6)
   02 ACCESSABILITY              CHARACTER(1)
        %This field is not used by the B1000
   02 RFS  %This field is reserved in the ANSII Standard.  It is
           %being used as follows by the B1000 and the B6700.
       03  MULTI.FILE.ID         CHARACTER(17)
             % "0" if there is no MFID
             % "X0" if Scratch
             % "BACKUP" if Backup
       03 SYS.SYMBOL             CHARACTER(2)
```

```
          % Will contain "17" if created on B1000
      03 TAPE.TYPE              CHARACTER(1)
          % 0 = Scratch
          % 1 = User
          % 2 = Backup
          % 3 = Library
      03 FILLER                 CHARACTER(6)
   02 OWNER.ID                  CHARACTER(14)
      % This field is not currently usable on the B1000 system
   02 FILLER                    CHARACTER(28)
   02 VERSION                   CHARACTER(1)
      % Will contain "1" until such time as the label format is
      % changed
```

The second of the three physical blocks is known as "Header One". The format is also used for End-of-File and End-of-Volume. A programmatic description is given below.

```
01 HEADER1.DECLARATION
   02 FILLER                    CHARACTER(4)
      % May contain "HDR1", "EOF1", or "EOV1"
   02 FILE.ID                   CHARACTER(17)
   02 FILE.SET.ID               CHARACTER(6)
      % This field will contain the first six characters from
      % the MFID field in the VOL1 block
   02 FILE.SECTION.NO           CHARACTER(4)
      % Used for Reel number by B6700 and B1000
   02 FILE.SEQ.NO               CHARACTER(4)
      % Ordinal number of the file within a Multi-File
   02 GENERATION.NO             CHARACTER(4) % Unused
   02 GENERATION.VERSION.NO     CHARACTER(2) % Unused
   02 CREATION.DATE             CHARACTER(6) % bYYDDD
   02 EXPIRATION.DATE           CHARACTER(6) % bYYDDD
   02 ACCESSABILITY             CHARACTER(1) % Unused
   02 BLOCK.COUNT               CHARACTER(6)
      % Zero if this is a Header.One block
   02 SYSTEM.CODE               CHARACTER(13) % "B1700"
   02 FILLER                    CHARACTER(7)
```

The third physical block is known as "Header Two". It is also used at End-of-File and End-of-Volume. Its format is shown below:

```
01 HEADER2.DECLARATION
   02 FILLER                    CHARACTER(4)
      % May contain "HDR2", "EOF2", or "EOV2"
   02 RECORD.FORMAT             CHARACTER(1)
      % F = Fixed
      % V = Variable
      % S = Spanned (Not yet implemented by any Burroughs system)
```

```
      % U = Undefined
   02 BLOCK.LENGTH              CHARACTER(5)
   02 RECORD.LENGTH             CHARACTER(5)
   02 RESV.SYSTEM.USE           CHARACTER(35)
      03 DENSITY                CHARACTER(1)
         % 0 = > 800
         % 1 = > 556
         % 2 = > 200
         % 3 = > 1600
      03 SENTINAL               CHARACTER(1) % Unused
      03 PARITY                 CHARACTER(1)
         % 0 = Even; 1 = Odd
      03 EXT.FORM               CHARACTER(1)
         % 0 = Unspecified
         % 1 = Binary
         % 2 = ASCII
         % 3 = BCL
         % 4 = EBCDIC
      03 FILLER                 CHARACTER(31)
   02 BUFFER.OFFSET             CHARACTER(2) % Unused
   02 FILLER                    CHARACTER(28)
```

As mentioned in a prior paragraph, the MCP writes ANSII Format labels on tapes whenever a file is opened output and the LABEL.TYPE field in the FPB is set to zero. If the user wishes to continue writing the old Burroughs format labels, he must modify this field in all of the files in his programs. This may be accomplished by recompilation, by the use of a File Attribute communicate operation within the program, by the use of the MODIFY control instruction or by the use of a FILE card when the program is executed. Presently valid values for the LABEL.TYPE field are:

0 = ANSII
1 = Unlabelled
2 = Burroughs

ANSII Labels, though they are written when the file is opened output, are actually created on all magnetic tapes prior to that time. A keyboard message has been implemented in the MCP for purposes of creating the initial ANSII label on all tapes. The mnemonic of the message is "SN" which used to be an acronym. for Serial Number. The syntax for SN is:

SN <unit mnemonic> <volume-identifier> ! ASCII !

<Volume identifier> may consist of one to six alphanumeric characters and is inserted in the VOLUME.ID field of the VOL1 block of the label which is created. This operation is, for conversational purposes, known as "initializing" the tape. All tapes and cassettes must be initialized on the B1000 before the

MCP will consider them scratch. This applies to seven-track, as well as all versions of nine-track tapes.

The <volume identifier> keyed in will remain on the tape until the tape is re-initialized. The tape may be purged at any time, provided the ANSII label is still intact on the tape. Tapes which have Burroughs labels on them must be re-initialized and may not be purged. Purging, here, implies the use of the "PG" keyboard message. Similarly, unlabelled tapes may not be purged, but may be re-initialied. The <volume identifier> is now part of the output of the "OL" message. The presence of the reserved word ASCII in an SN statement causes the label to be written in ASCII character codes.

The capability of creating and recognizing ANSII labels was not included in the MCP prior to the 5.0 release of the software. Before the 5.0 release, all labels created by the B1000 system were the old Burroughs labels first implemented on the B5500 system. A programmatic description of these labels, as they are created on the B1000, is shown below. As can be seen from the description, certain fields have been added to the labels to improve their utility. These fields are meaningful to the B1000 system only. A programmatic description is presented below.

```
DEFINE STANDARD.LABEL.DECLARATION AS # %
DECLARE 01 DUMMY REMAPS L.LABEL.RECORD %
,    02 L.LABEL      CHAR (9)  %        " LABEL  0"
,    02 L.MFID       CHAR (7)  %                  "
,    02 L.Z1         CHAR (1)  %        "0"       "
,    02 L.ID         CHAR (7)  %
,    02 L.REEL       CHAR (3)  %
,    02 L.DW         CHAR (5)  %        DATE WRITTEN
,    02 L.CYCLE      CHAR (2)  %        "0"
,    02 L.PID        CHAR (5)  %        PURGE DATE
,    02 L.S          CHAR (1)  %        SENTINNEL (1 = END-OF-REEL)
,    02 L.BC         CHAR (5)  %        BLOCK COUNT
,    02 L.RC         CHAR (7)  %        RECORD COUNT
,    02 L.PB         CHAR (1)  %        PRINT BACKUP FLAG
,    02 L.SERIAL     CHAR (5)  %        SERIAL NUMBER
,    02 L.SYSTEM     CHAR (5)  %        CREATING SYSTEM
,    02 L.BUFSIZE    CHAR(8)   %        NEW FORMAT DECIMAL BLOCK SIZE
,      03 L.BSIZE    BIT(24)   %        OLD FORMAT BINARY
,      03 L.RSIZE    BIT(24)   %        OLD FORMAT BINARY
,    02 L.RECSIZE    CHAR(8)   %        NEW FORMAT DECIMAL RECORD SIZE
,    02 L.MODE       CHAR(1)   %        NEW FORMAT RECORDING MODE FOR
                               %        TAPE FILE
;#
```

All labels on the B1000 system are written in odd parity. Beginning with the 4.2 release of the software, tape marks are

written in even parity, except where prohibited by the control. This was done as an accomodation to the B300 system, which can read only seven-track tape and cannot recognize tape marks which are written in odd parity.

MCPII will write tapemarks and ending labels on any output labeled tape that is not at BOT when a Clear/Start is done. This will allow the user to read that tape and recover the data. There is one restriction. If the tape is to be read in reverse, the user must specify blocking information.

ANSII labels are also written as the standard label on seven-track tape. When this is done, the labels are written with translation to BCL. Burroughs labels, when written to seven-track tape, are written in odd parity with the EBCDIC/BCL translator enabled.

The STATUS Procedure makes all possible attempts to recognize a label when a tape unit becomes Ready. On seven-track tape, particularly, there are several different variations of parity and recording mode that may have been used to create the tape. Seven-track tape can be written with or without character translation from EBCDIC to BCL. The MCP will attempt to read tape labels with all possible variations before giving up.

When the MCP cannot recognize a label, the unit is considered available for input purposes if the tape does not have a Write Ring in it. In this case, it must be manually assigned to a program by the operator, either when the program requests the file or when the job is executed. If the tape does contain a Write Ring, it must be initialized, using the SN instruction decsribed above. Only when the tape has a Write Ring and contains a valid ANSI label indicating "Scratch" is it considered available for output purposes automatically by the MCP.

It is also the responsibility of the STATUS Procedure to record the other information returned by the Test I/O operation. This information is crucial to the proper operation of the tape subsystem. In particular, if the system is equipped with a PE/NRZ exchange, the operation of the STATUS Procedure when a unit becomes Ready is as described below.

PE/NRZ EXCHANGES

With the inclusion of the M4/M5 MEC supplied by the Westlake Plant and described by P.S. #2047 4490, it is possible for a tape unit to operate in either Phase Encoded (PE) or Non-Return to Zero (NRZ) recording mode. This can only be accomplished on the B1000 hardware by connecting one NRZ control and one PE control to the MEC. The NRZ control is designated MTC-2 and the PE control is designated MTC-4. A tape subsystem so connected is spoken of as an exchange subsystem by hardware personnel. According to the software definition of a subsystem, all controls in the subsystem must be identical. The code in the I/O driver which interfaces to MTC-2 is distinctly different from that which interfaces with MTC-4. A request for a unit which is operating in the NRZ mode can only be handled by MTC-2.

To solve this problem, considerable coding has been incorporated in the MCP. The problem has been rectified in the most efficient manner possible, however. Two separate chains of descriptors, one for each control, are constructed by the MCP at Clear/Start time. The two chains are maintained by the MCP dynamically, from that point.

Recording mode information is supplied by the test operator and actually is returned as the density field in the result descriptor. A density selection of 1600 bpi, for example, indicates that the unit has been selected to be in the phase-encoded recording mode and that the I/O descriptors for the unit should be in the MTC-4 chain of descriptors. If the subchain for the unit is not in the proper chain, the MCP will move the entire subchain to the proper chain. The movement of the subchain is only attempted when the unit is not in use, of course. Selecting a different density while the unit is being used constitutes an error on the part of the operator. The operator is notified of the error and the program is allowed to continue processing only when the proper density has been selected on the unit.

This solution is only possible if both controls are capable of reporting recording density properly. MTC-2 can report the fact that a unit is selected to be in the 1600 bpi density. Similarly, MTC-4 is able to report the fact that a unit is in the 800 bpi density. Density information is commonly used by the MCP only when a unit goes from a not-ready state to a ready state. The movement of the subchain is therefore performed by the MCP status routine when the unit becomes ready.

Unit mnemonics are not affected by the presence of a PE/NRZ exchange. A unit selected as MTA, for example, will always be

known as MTA, regardless of which chain contains its subchain, or which density is selected by the operator.

Due to differences in the unit numbering scheme between MTC-2 and MTC-4, there can be no more than eight magnetic tape units connected to a PE/NRZ tape subsystem. This capability is not available on any version of the software prior to the 5.1 release version.

FILE STRUCTURES

A File is a group of related records. Files are of central importance in the I/O Subsystem since effectively all of the communication between user programs and the subsystem is accomplished through files.

The B1000 Operating System supports three different file types or structures, exclusive of Data Management System structures, which correspond roughly to those file types defined in the ANSI '74 COBOL Language. In that language, these types are called Sequential, Relative and Indexed Sequential. Sequential and Indexed Sequential files, in COBOL, can both be accessed in a random manner and the use of the word "Sequential" tends to add confusion. In this document, the three types will be refferred to as Conventional Files, Relative Files and Indexed Files.

CONVENTIONAL FILES

The basic definition of Conventional file structures is found in the COBOL '68 Language, though many functions have been added to the basic definition. To a program, a file represents a large collection of ordered data that exists apart from the program. The program needs to interact with parts of that data from time to time and the I/O Subsystem makes this interaction possible. The I/O Subsystem moves the data into and out of user working areas in main memory, to which the program has access.

The unit of data moved into and out of the user's working area is the record. The record is considered, by the I/O Subsystem, to be a string of bits, which the user program will probably group into characters or words in some manner, but the I/O Subsystem deals only with entire records and delivers and receives one record at a time to and from the user program.

A file has some structure as seen by the user program. The records may be all of the same length or they may be of variable length. Length information must be declared by the program or contained in the record itself or exist in an accessible form in the physical file or exist in the information which the MCP maintains about the file. If the record length is variable, then the length of each record must exist in that record, in the first four character positions.

The file, as it is stored on some recording medium, is often refferred to as a physical file. A physical file may have some additional elements of structure. It may contain blocks. A block is a group of physically contiguous records which are transferred to and from the physical medium as a group. The storage device itself may impose some structure upon the file. As discussd previously, data is transferred to disk in 1440-bit increments. A block of records to be written to disk must therefore total some integer multiple of 1440 bits. The disk itself may be used to store many disjoint physical files. To minimize storage availability problems, the MCP allows disk files to be broken into "areas", each of which will contain room for a specified number of blocks. This is described in more detail later.

The physical file inherits many of its properties from the logical file declared by the user program which creates it. When the user programmer declares a logical file, the compiler generates a File Parameter Block which contains the specified values for the various attributes of the file. File Parameter Blocks (FPBs) are defined in Section 2 of this specification. The MCP, and more specifically the OPEN procedure, converts the attributes specified by the user to an actual physical file. More attributes are added to the physical file when it is assigned to a device.

Any file may be described by its attributes. File attributes are system control parameters which are used by the I/O Subsystem. The attributes contain all of the information the subsystem needs when it connects a physical file to a logical file declared in a user program and when it controls the access to that physical file.

Most of the attributes associated with any file are contained in the File Parameter Block (FPB) for that file. Certainly, the FPB is the storage medium for the attributes that are declared by the user and generated by the compiler. Additional attributes will be obtained when the file is opened and assigned to a device. When a file is open, its attributes may be stored in the FPB, the File Information Block (FIB), the Disk File Header (DFH) and the I/O Assignmment Table (IOAT). All of these structures have been presented previously.

Beginning with the 8.0 version of the MCP, a communicate operation was added to allow user programs to dynamically modify selected attributes of a file. In subsequent versions of the MCP, the list of modifiable attribtes has been expanded. The File Attribute communicate operation is described in the Demand Management section of this document.

FILE NAMING CONVENTIONS

All names associated with files on the B1000 MCP may be a maximum of ten characters in length. Names in excess of ten characters will be truncated to the first ten. Looking at the description of the FPB presented in Section 2 of this specification, the first field in the FPB, FPB.FILE.NAME is the internal name of the file. "Internal", in this case, means internal to the user program. This is the name which appears in the File Declaration of the user program and the name which the programmer uses in all references to the file within the program.

The next three name fields in the FPB provide the "File Identifier" for MCP purposes. All physical files introduced to the system may have one or two names. Files assigned to disk pack may have a third name which will correspond to the pack name, the name contained in the pack label.

If a file has one name only, that name is stored in the field FPB.MULTI.FILE.ID and the field FPB.FILE.ID should be filled with blanks. FPB.MULTI.FILE.ID is often referred to as the "Family ID" and is only important if the file is assigned to disk or tape. If a file has two names, the second name is stored in the FPB.FILE.ID field.

The assignment of physical files to logical files is discussed in the Demand Management Section of this specification in the description of the OPEN communicate operation. Stated in its simplest form, the MCP attempts to associate one or two names with each device that is connected to the system and that is capable of input operations and to match this external name to the File Identifier specified in an FPB when a user OPENs a file. On output files, the MCP simply attempts to assign an available device of the requested hardware type.

There are two exceptions to the statements in the preceding paragraph. When an output file is directed to Printer or Punch devices, the output data may be actually stored on disk for later retrieval. Such files are known as Backup Files and are discussed later. Input card files may be loaded to disk files prior to the time they are required by a program. When the program then requests the card file, MCP may automatically substitute the previously loaded disk files. This is known as the Psuedo-Reader facility and is discussed in Product Specification 2222 2265, SYSTEM/LDCNTRL.

LOGICAL DISK FILES

It is the MCP's responsibiltiy to convert a logical disk file as declared in a user program, to an actual physical disk file. This can only occur by a program opening a new disk file, where "new" in this context specifies that the program intends to create a file and the physical disk files that are currently known to the system are of no concern to the user.

Except in the case of Multi-Pack files, files that extend over more than one physical pack or cartridge, a new file can only become a permanent file that exists when the program is no longer executing by the same user doing a close operation on the file and specifying in the CLOSE communicate operator that the file is to become permanent. This implies that the file identifier is to be entered in the disk directory and remembered by the MCP forever. This also implies that the disk storage space occupied by that file is to be used for no other purpose except the various user manipulations that may occur within that file, utilizing a logical file with the same File Identifier. The Close operation is also described in detail in the Demand Management section of this specification. Basically, the Open and Close operations both obey the rules presented in the definition of the COBOL Language.

PHYSICAL DISK FILES

In order to manage all of the available storage space on a disk device, the MCP must maintain tables which tell it the storage locations that are available for use, the names of the files that are already stored on the disk and the physical characteristics of those files.

DISK SPACE ALLOCAIION

There are three tables, each with the same format, that are used by the MCP to allocate disk space. The master available table is a non-expandable table of three contiguous segments beginning at the second sector on disk. It contains a list of all unusable segments which have been "XD-ed" by the operator. The working available table is a 10-segment table beginning at the 47th disk segment. It contains a list of all available or unused space on disk and is expandable as needed. The temporary table is five contiguous segments and contains a list of all segments in use but not reflected in the disk directory. This expandable table begins at the 57th sector. At Clear/Start time, all sectors in the temporary table are returned to the available table. A programmatic description is given below:

```
    DEFINE
    DISK.AVAILABLE.DECLARATION AS#
DECLARE
    01 DUMMY REMAPS DISK.AVAILABLE BIT(SEG.SIZE),
       02 AVL.FOR.LINK        DSK.ADR,
       02 AVL.BACK.LINK       DSK.ADR,
       02 AVL.SELF            DSK.ADR,
       02 FILLER              BIT(4),
       02 AVL.BLOCK(22),
          03 AVL.ADDRESS      DSK.ADR,
          03 AVL.LENGTH       WORD;#;
```

FILE ACCESS AND IDENTIFICATION

The disk directory is the structure which catalogues and points to all files on disk. Each entry contains the file's name, type, and Disk File Header (DFH) address. The directory is a two-level structure containing a primary or "master" directory and a secondary directory. The master directory is created at Cold Start as 16 contiguous disk sectors beginning at sector 31. Each sector contains entries for eleven files. As each sector is filled, another disk segment is allocated and linked to the filled sector. If a file has two names, the primary name (Multi-File IDentification) is placed in the master directory with a pointer to a secondary directory, where all the files with that MFID are listed. The secondary directory is structured and linked in the same fashion as the master directory. A programmatic description is given below:

```
DECLARE 01 DIRECTORY REMAPS BASE,
           02  DISK.SUCCESSOR          DSK.ADR,
           02  DISK.PREDECESSOR        DSK.ADR,
           02  DISK.SELF               DSK.ADR,
           02  FILLER                  BIT (12),
           02  DISK.NAME               NAME,
           02  DISK.ADDRESS            DSK.ADR,
           02  DISK.FILE.TYPE          BIT (4),
           02  FILLER                  BIT (1200); % 11 ENTRY PER SEG
```

The Disk File Header (DFH) is a variable-length header record, the size of which is dependent upon the number of declared areas in the file and is computed as follows:

540-BITS + (36-BITS * NUMBER-OF-AREAS)

The DFH is never less than 1440 bits nor greater than 4320 bits on disk. It lists the physical characteristics of the file including its file type and the disk address for each area. The following file types are recognized by the MCP:

LOG
DIRECTORY
CONTROL DECK
BACKUP PRINT
BACKUP PUNCH
DUMPFILE
INTERPRETER
CODE FILE
DATA FILE
VARIABLE LENGTH RECORD DATA FILE
INTRINSIC FILE

DISK FILE IDENTIFICATION

As discussed previously, Disk File Headers (DFH) are the structures used to identify a file on disk. It is a variable-length record which describes the physical attributes of the file and contains pointers to each "area" of the file. When a disk file is "opened", a copy of the DFH is copied into memory. The header in memory points to the header on disk and vice versa. There will never be more than one copy of the header for a file in memory at any time. Multiple users of the file will use the same copy of the header. Maintenance of disk file headers is covered in another section. A programmatic description is given below:

DISK FILE HEADER

```
DEFINE FILE.HEADER.DECLARATION AS #%
    FH.MAP(FILE.HEADER)#,
FH.MAP(FILE.HEADER) AS #%
DECLARE 01 DUMMY REMAPS FILE.HEADER,%
    02  FH.USERS.RANDOM       BIT(8),% FORMERLY FH.CORE.ADDR
    02  FH.NEWFILE            BIT(1),% CLEARED WHEN NEW FILE IS FILED.
    02 FILLER                 BIT(7),
    02  FH.FILE.KIND          BIT(8),
    02   FH.SELF                    DSK.ADR,
    02   FH.NO.USERS                BIT (8),
    02   FH.USERS.OPEN.OUT          BIT (4),
    02   FH.OPEN.TYPE               BIT (4),
    02   FH.FILE.TYPE               BIT (4),
    02   FH.PERMANENT               BIT (4),
    02   FH.JOB.WAITING.ON.CLOSE    BOOLEAN,
    02   FILLER                BIT(9), % DON'T USE UNTILL 1977
```

```
    02  FH.HDR.SIZE            BIT(14),% LENGTH OF MYSELF IN BITS.
    02  FH.NO.USERS.LOCK             BIT(4),
              % NO.USERS WHO HAVE IT OPENED WITH LOCK
    02  FH.RECORD.SIZE         BIT(20),% LENGTH IN BITS.
    02  FILLER                 BIT(4),% DON'T USE TILL 1977
    02  FH.RCDS.BLOCK          BIT(20),%
    02   FH.BLOCKS.AREA              WORD,
    02   FH.SEGS.AREA                WORD,
    02   FH.AREAS.RQST               BIT (12),
    02   FH.AREA.CTR                 BIT (12),
    02   FH.EOF.POINTER              WORD,
    02  FILLER                 BIT(4),%DON'T USE TILL 1977
    02  FH.BPS.NO              BIT(20),%
    02  FH.BLOCK.COUNT         BIT(24),% DON'T USE TILL 1977. IGNORED 5.1.
    02  FH.FORMAT              BIT(3),% HITHERTO =0.  FOR RELEASE, =1.
    02  FH.MPF                 BIT(1),% HITHERTO 4 BITS.
    02  FILLER                 BIT(24),
    02  FH.CREATE.TIME         BIT(16),% HITHERTO 0. HENIGE'S GENEROSITY.
    02  FILLER                 BIT(8),
    02   FH.USER.INFO                WORD,
    02   FH.SAVE.FACTOR              BIT (12),
    02   FH.CREATION.DATE            BIT (16),
    02  FH.ACCESS.DATE         BIT(16),%
    02  FH.SER.NO              BIT(24),% DON'T REUSE TILL 1977. 5.1 IGNORE
    02   FH.MPF.ADDR            DSK.ADR,  % DONT REUSE TILL 1977
    02   FILLER                      BIT(1),
    02   FH.UPDATE.VERSION           BOOLEAN,
    02   FH.DMS.WRITE.CONTROL,
         03   FH.DMS.TO.BE.WRITTEN       BOOLEAN,
         03   FH.DMS.CONTROLPOINT        BOOLEAN,
    02   FH.VERSION              BIT(36),       % YEAR,JDAY,TIME
    02 FH.PROTECTION                  BIT (2),%  HOST RJE
    02 FH.PROTECTION.IO               BIT (2),%  HOST RJE
    02 FILLER                         BIT (16),% HOST RJE
    02   FH.AREA.ADDRESS (105)  DSK.ADR,
      03 FH.UNIT                 BIT (12), %
         04 FH.PORT              BIT (3), %
         04 FH.CHAN              BIT (4), %
         04 FH.SER.NO.FLAG       BOOLEAN, %
         04 FH.EU                BIT (4), %
         03   FH.ADDR            BIT (24);
#;%
```

MULTI-PACK FILES

The B1000 MCP includes the capability to allow a file to extend over more than one removable pack or cartridge. Such a file is known as a "Multi-Pack File" (MPF). Quite obviously, there are some limitations on the use of such files. The individual packs or cartridges which contain portions of the file may not be removed indiscriminately. Various operational details are contained in the "B1700 Software Operational Guide".

BASE PACKS

A multi-pack file may have only one "Base Pack" (BP). The name of the base pack is the pack id as specified by the user in the FPB of the multi-pack file. The base pack must be on line for all OPENs of the file. The MCP may also require that the base pack be on-line for other operations, such as the assignment of a new area of disk to the file. An appropriate message will be typed on the console printer by the MCP if the base pack is required and it is not on-line. The operator may then mount the base pack and the requesting program will continue. The base pack must be on line when the file is closed if it was opened for output or input/output.

A base pack may contain single files, as well as multi-pack files, in any combination. It may not, be a "continuation pack" for a multi-pack file whose base pack is a different physical pack or cartridge.

The file header for a multi-pack file is contained on the base pack. It contains all information concerning the file, including the addresses of every area assigned on the base pack to that file. For each area which resides on a continuation pack, the header will contain the serial number of the continuation pack. This allows the MCP to control all processing of the file and thereby avoids the necessity of updating each continuation pack as the file is processed.

CONTINUATION PACKS

A multi-pack file may, by definition, reside on two or more packs or cartridges. When the file overflows or "continues" to additional packs, the term "continuation pack" is used. A multi-pack file may reside on up to sixteen packs or cartridges. There may be up to fifteen continuation packs assigned to one multi-pack file.

A continuation pack may be associated with only one base pack. A continuation pack may contain only continuation files; it may not be a base pack for another file. A continuation pack may contain information associated with more than one multi-pack file, but all of the files must be assigned to the same base pack.

The file header, which is contained on the base pack for a multi-pack file, contains disk addresses for only those areas of disk which are assigned to the base pack. The same statement can be made of continutation packs; the file header contained on a continuation pack contains disk addresses that are assigned on that pack only. The file header on the base pack contains the serial number of the appropriate continuation pack in the disk address fields of the headers.

When a file overflows from the base pack, the MCP will search for another continuation pack that is already on-line and that is associated with the same base pack. If such a continuation pack is found, the file automatically overflows to that continuation pack. If no such continuation pack is present on the system, the MCP will then search for a scratch pack, one which has no files on it, with the same type as the base pack. "Type" here means "restricted" or unrestricted" and is determined when the pack is initialized.

If such a scratch pack is found, the file automatically continues to that pack. If no such pack is found, the MCP temporarily halts the program and prints an appropriate message on the console printer. The program may be continued when a suitable continuation pack is present on the system.

MULTI-PACK FILE INFORMATION TABLE

When a multi-pack file is opened input, the file's header is read into memory from the base pack. When a multi-pack file is opened output, and new, a header is constructed in memory from information in the program's FPB and information from the base pack. During OPEN the MCP will find space on the system pack for a multi-pack file information table. The table will contain specific information about the base pack, along with an exact copy of the disk file header from the base pack. This copy of the header is treated as a working copy while the file is open. The header on the base pack may therefore not always be correct.

The format of the MPF.INFO.TABLE is presented below. One MPF.INFO.TABLE per file is required, regardless of the number of users.

FIELD NAME	TYPE	DESCRIPTION
01 MPF.INFO.TABLE	1392 BITS	
02 MPF.FORWARD	36 BITS	POINTER TO NEXT MPF TABLE.
02 MPF.BACKWARD	36 BITS	POINTER TO PREVIOUS MPF TABLE.
02 MPF.SELF	36 BITS	POINTER TO THIS MPF TABLE.
02 MPF.NAME	30 CHAR	FILE-IDENTIFIER.
02 MPF.HEADER.SIZE	24 BITS	SIZE OF COMPOSITE HEADER MAINTAINED BY THE MCP.
02 MPF.HEADER.ADDRESS	24 BITS	POINTER TO THE COMPOSITE HEADER IN MEMORY.
02 MPF.BPS.NO	24 BITS	BASE PACK (BP) SERIAL NUMBER.
02 MPF.OPEN.TYPE	4 BITS	TYPE OF FILE OPENED. SAME AS DFH.OPEN.TYPE IN DISK FILE HEADER.
02 MPF.NEW.FILE	1 BIT	MCP FLAG USED IF THIS IS A NEW FILE.
02 MPF.NEW.AREA	1 BIT	MCP FLAG USED IF NEW AREA WAS ADDED.
02 MPF.CS	BIT	MCP FLAG TO MARK IF CLEAR/START WAS PERFORMED SINCE THIS ENTRY WAS CREATED.
02 FILLER	1 BIT	
02 MPF.BASE.PACK.TYPE	4 BITS	TYPE OF PACK USED AS BP. 1=RESTRICTED, 2=UNRESTRICTED
02 MPF.ARRAY		USED TO RECORD ALL PACKS THAT ARE ON-LINE.
03 MPF.ONLINE		MAXIMUM OF 16 ITEMS IN ARRAY.
04 MPF.SERIAL.NO	24 BITS	SERIAL NUMBER OF THE PACK.
04 MPF.HDR.DSK	36 BITS	DISK ADDRESS OF THE FILE HEADER ON THE PACK.

MULTI-PACK FILE GENERAL RESTRICTIONS

In addition to any restrictions listed in the foregoing, the items below are also applicable to multi-pack files.

1. Since a system cartridge may not be a base pack, multi-pack files are only operational on systems with two or more drives.

2. All packs containing any part of a multi-pack file must have unique serial numbers.

PRINTER FILES

All Burroughs printers and controls have hardware capability of spacing the paper after writing a line of output but no capability of spacing the paper before writing the line. With the advent of the ANSI '74 COBOL Language in the 9.0 version of the software, the need for a more efficient means of performing the COBOL WRITE AFTER ADVANCING statement became apparent. In prior versions, this operation was implemented by the compilers, generated two actual I/O communicate operators for each such statement encountered. The first of the two was a Position communicate or a WRITE of a line of blanks; the second was a WRITE of the actual record with no paper motion specified. This, of course, resulted in two communicates as well as two physical IOs for every logical WRITE AFTER ADVANCING operation. The change described below was first implemented in the 9.0 Operating System and is included in all subsequent versions.

The goal of this modification was to reduce the number of communicate operations to one per logical WRITE and to reduce the physical I/O operations to one per communicate operation using the existing printer hardware. This was accomplished by delaying the initiation of the physical I/O operation until the following logical WRITE is received. By knowing both the previous and current logical I/O requests, a physical I/O can be initiated which corresponds to the first request and takes advantage of the Burroughs hardware.

The diagram in Figure 1 shows the relationship between the last logical request issued by the user, the current logical request and the actual physical I/O operation that will be performed.

Current Logical Request \ Pending Operation	Null	Write, No Space	Write Before Single Space
Write, No Space	No-op, Pending:= Write/No	Write, No Space Pending:= Write/No	Write, Space 1 Pending:= Write/No
Write/B Space 1	No-op, Pending:= Write/B Space 1	Write, No Space Pending:= Write/B Space 1	Write/B Space 1 Pending:= Write/B Space 1
Write/B Space 2	Write/B Space 2 Pending:=Null	Write, No Space Write/B Space 2 Pending:=Null	Write/B Space 1 Write/B Space 2 Pending:=Null
Write/A Space 1	Space 1 Pending:= Write, No Space	Write/B Space 1 Pending:= Write, No Space	Write/B Space 2 Pending:= Write, No Space
Write/A Space 2	Space 2 Pending:= Write, No Space	Write/B Space 2 Pending:= Write, No Space	Write/B Space 2 Space 1 Pending:= Write, No Space
Write/B Channel	Write/B Channel Pending:=Null	Write, No Space Write/B Channel Pending:=Null	Write/B Space 1 Write/B Channel Pending:=Null
Write/A Channel	Space Channel Pending:= Write, No Space	Write/B Channel Pending:= Write, No Space	Write/B Space 1 Space Channel Pending:= Write, No Space
Space N	Space x Space (N-x) Pending:=Null	Write/B Space x Space (N-x) Pending:=Null	Write/B Space 2 Space (N-1) Pending:=Null

Figure 1 - Logical/Physical I/O Relationship

In the preceeding diagram, the operations within the table correspond to the actual physical I/O operations that will be performed, which will depend upon the current logical request supplied by the user and any operations that are still pending from the previous request. Write/B and Write/A may be read "Write Before" and "Write After". The symbol (:=) may be read "is replaced by". It can be seen in the diagram that some logical requests will, at times, result in two physical

operations being initiated. Under these conditions, it may be beneficial to supply each printer file with at least two buffers, if the execution time of the program is the only concern. Total system throughput will not be impacted significantly regardless of the number of printer buffers and the types of operations being performed. If the MCP must wait for the completion of any printer physical I/O operation, the time that is spent waiting will be masked by the processing of other programs.

Along these same lines, it should be remembered that any time a Write operation is left pending and control is returned to the user, the MCP must have an available buffer to store the data that is to be written. If no buffer is available, control may not be returned to the requesting user until a buffer becomes available. Again, this time will be overlapped with the processing of other programs and system throughput should not be significantly impacted.

The action presented in the preceeding chart for a Space operation requires some explanation. A Space of more than two lines must be handled by the S.MCP. The Micro MCP will attempt to space the requested number of lines without calling the S.MCP, but this is not always possible. In the diagram, when the Pending operation is equal to Null, the Micro MCP will space the paper one or two lines, indicated by "x" in the diagram, and if N-x is greater than zero, it will pass the remainder to the S.MCP. Similarly, when the Pending operation is equal to a Write with No Space, the Micro MCP will issue a Write/B Space 1 or 2 lines, also indicated by "x" in the diagram, and if the remainder is greater than zero, pass it to the S.MCP. When the Pending operation is a Write/B Space 1, the Micro MCP will issue a Write/B Space 2 and pass N-1 to the S.MCP, if N-1 > 0.

LINAGE Clause

The LINAGE clause, in ANSI '74 COBOL is a mechanism which allows the user to define a "Logical Page" format and request that the Operating System maintain printer pages which conform to the defined format, as well as a current line position on that logical page. In the language, the user may specify the Logical Page size, an integer which represents the number of lines that may be printed on any page. This attribute will be known as PAGE.SIZE in the remainder of this discusion.

The user may also specify an Upper Margin, an area at the top of each page where nothing will be printed, Lower Margin, a similar area at the bottom of each page, and a Footing area, a specified number of lines in the page body immediately above the Lower Margin area. The user may also ask to know the number of the

line in the page body where the last line of output was printed. This requires that the Operating System maintain a line counter, which will be the number of lines written on the current page.

The implementation is called the "Logical Page" function in the Operating System and it includes the following:

1. Positioning to the beginning of the page body i.e. past the top margin at OPEN or at page overflow.

2. Reporting End-of-Page when the user writes or spaces within the footing area and requests EOP reporting.

3. Detecting page overflow. Page overflow is defined as occurring whenever the execution of a WRITE would leave the line counter positioned past the page body.

4. Updating the logical page description when switching from one logical page size to another.

Essentially, the implementation obeys the rules presented in the ANSI '74 COBOL specifications. The operating system will maintain a line counter, a current logical page description and a new logical page decription. The line counter represents the position on the page body following the open or the last logical write. The current logical page description is used to detect end-of-page and page overflow. The new logical page description is used to initialize the current logical page description when page overflow is detected and to calculate the number of lines to the first line of the next page body.

If the user has specified end-of-page reporting and the line counter is greater than or equal to the line number at which the footing begins, then on completion of the WRITE, EOP is reported to the user. If the line counter would be greater than the line number at which the bottom margin begins at the end of the logical WRITE, an implicit position to the first line of the next page body is generated according to the before/after variant of the write statement. At this point the line counter will be set to 1. The number of lines to skip is calculated according to the following formula:

```
lines.to.skip := current.page.body.size - line.counter +
   current.bottom.margin.size + new.top.margin.size;
```

The Logical Page description is updated if necessary when a write occurs that causes page overflow or whenever an advance to top of page occurs.

To access the line counter requires a File Attribute Communicate from the user program. This will be of no concern to ANSI '74 COBOL users; they need only be concerned with the proper syntax in that language for referencing the line counter. The Logical Page definition is changed to the values included in the Write Communicate format whenever page overflow is detected. To accomodate the above requirements, the format had to be expanded as shown in Figure 2 in the WRITE AFTER ADVANCING section of this document presented previously.

The Logical Page implementation, since it is implemented entirely in software, is useable even when the file is directed to a Backup medium. The Logical Page implementation is also useable by programs that are written in languages other than ANSI '74 COBOL. This is effected by the implementation of additional syntax in the FILE Control Card. Progarms may be permanently modified to incorporate the required new attributes. The Logical Page function is activated by the PAGE.SIZE attribute in the File Parameter Block. When a printer file is opened and PAGE.SIZE contains a value other than zero, page format will be controlled by the Logical Page software implementation and the physical carriage control tape on the device will be completely ignored after the file is open.

It is important to note that the Channel One punch, as well as the Channel Twelve punch in the carriage control tape is ignored after the file is open. According to ANSI '74 COBOL specifications, this is as it should be but it dictates that the attributes which govern logical page format must be specified such that the logical page size plus the upper margin plus the lower margin must total the exact number of lines on the physical page. If this is not done, then eventually at least, lines will be printed on the crease between the physical pages.

The relevant attributes may be referenced in the FILE Control Card as shown below.

Attribute	Abbreviation	Function
PAGE.SIZE	P.S	The number of lines between the Upper Margin and the Lower Margin. May be set to any value between 1 and 255 inclusive.
LOWER.MARGIN	L.M	The number of lines from the page body to the bottom of the form. May be set to any value between 0 and 255 inclusive.
UPPER.MARGIN	U.M	The number of lines from the bottom (or top) of the form to the page body. May be set to any value between 0 and 255 inclusive.
FOOTING	FOOT	The number of lines from the beginning of the page body, within Page.size, to the point where the MCP will begin to report end-of-page to the user. May be set to any value between 1 and 255 inclusive.

PRINTER AND PUNCH BACKUP CAPABILITIES

The MCP includes the capability of directing the output data for printer and punch files to intermediate storage. The storage medium may, at the user's option, be magnetic tape or disk. Backup files may not be directed to cassette or flexidisk media. A utility routine, named SYSTEM/BACKUP, is provided to allow users to retrieve the output data from the intermediate storage medium. For details on this routine, refer to Product Specification 2222 2681, System Backup.

When the output is directed to magnetic tape, multi-file tapes are created unless the operator intervenes in some manner. If the operator does not intervene, the tape will be closed with no rewind when the printer or punch file is closed in the program. The next printer or punch file which is opened by any executing program and directed to backup tape storage will then be added to the existing tape. This process will continue until the operator intervenes or until the physical end of the tape reel is reached. Operator intervention procedures are described in the Software

Operational Guide and in the MCP Control Syntax Product Specification.

When the output data is directed to intermediate storage on disk, it is entered in the Disk Directory when the printer or punch file in the program is closed. At that time, it may be accessed by any program, though the data contained therein may be undecipherable unless the accessing program is written expressly for this purpose. The file may not, under any circumstances, be accessed prior to the time the file is closed.

BACKUP FILE BLOCKING FACTORS

The OPEN routine in the MCP attempts to optimize the size of the physical blocks associated with a Backup file, according to the declared size of the logical records in the file. The block will typically be set to a size equivalent to three or four disk sectors, each of 180 bytes, by the MCP. In order to predict the block size that the MCP will select for any given logical record size, it is necessary to consider the control information that the MCP stores in the first physical block of the file as well as the declared record size. The algorithm that is used by the MCP to select a block size is not easily described. The block size which is selected is stored in the file label, for tape files, and in the Disk File Header for disk files. The logical record size is also stored in these fields.

Consequently, using the Default File Attribute, which is described in the Software Operational Guide and in another part of this specification, the user may access Backup files without knowing the blocking factor and logical record size in advance. Since the algorithm that is used by the MCP to calculate block size may change from version to version, this means of determining the blocking factor used is preferred. The algorithm that is included in the 8.0 version of the MCP is described in the paragraphs that follow.

The logical record size declared in a file in a user's program may be any size. If the file is directed to Backup storage, it is set to a maximum of 132 bytes. The logical record size is then incremented by two bytes. This additional sixteen bits of information is necessary to contain the formatting information which is passed with each Write and Position communicate operator.

If the file is being directed to magnetic tape, the record size is then incremented, if necessary, to force it to a number which is modulo forty-eight. This is necessary since seven-track tape

units require block sizes which are modulo six and phase-encoded drives require block sizes which are modulo sixteen. It would not be sufficient to insure that only block sizes meet this requirement, however, since the blocks on any tape file may be partial blocks which contain one or more records.

The buffer size will always be made large enough to contain 100 bits of control information plus 1668 bits to contain the original File Parameter Block as it appeared in the user's program, plus, if the file is a printer file, 1072 bits to contain a file label plus its associated spacing information. If the original file is a punch file, a space of 648 bits is reserved for the label instead of 1072. The one fact which complicates this calculation is that all three of the items listed above must begin on a logical record boundary within the physical block. Consequently, for a file with a declared record size of 132 bytes, which is converted to 134 bytes or 1072 bits by the OPEN routine, the FPB will begin on the 1073rd bit in the first physical block of the file. The file label, if there is one, will begin on the 3217th bit (3 x 1072). The first output data record will then begin on the 4289th bit. The block will be made large enough to insure that the first block contains at least one logical record in addition to all of the information listed above.

For backup files which are directed to intermediate storage on disk, the block size computed above is then incremented, if necessary, to make the size module 1440. The number of records per block is then computed from record size and block size. End-of-File is never reported to a user program when a Backup file is being created. The MCP automatically closes the file when it is full and also automatically opens a new Backup file. The identifier assigned to the second file will revert to the standard naming convention for Backup files. The MFID will be set to BACKUP.PRT and the ID field will be set to the next sequential number maintained by the system. All other Backup file attributes, such as the number of copies requested, will be retained in the second and subsequent files. Only the name requested by the user will be lost.

The MCP also allows users to specify the file attributes Blocks per Area (BLOCKS.AREA or B.A), Records per Block (RECORDS.BLOCK or R.B), and Number of Areas (AREAS or ARE) for printer files and these specified values will override the system's default values for the same attributes. Using the proper setting of these values and the automatic closing and reopening described in the preceeding paragraph, users may begin printing a Backup file while the program which created it is still executing and creating the second or subsequent portion of the same file.

Records in Printer files may not be blocked. Consequently, the Records per Block attribute is not applicable when the file is directed to the printer. Records per Block is utilized only when the file is directed to a Backup medium. Also, the value specified for Records per Block must be greater than a minimum value, which is a function of the record size associated with the file and which is computed by the MCP when the file is opened. It is reccomended that users not set Records per Block for Printer files in the use of this facility but establish the file size via the Blocks per Area and Number of Areas attributes only. For a file with 132-byte records, Records per Block will be set to five by the MCP unless overridden by the user. The simplest means of determining the value that will be computed for Records per Block by the MCP for any other given record size is to direct such a file to the backup medium and interrogate Records per Block.

The MCP insures that access to a backup file is in serial mode only. If the user had requested more than two buffers on the original file, the number is reduced to two on the backup file. In a similar manner, the MCP limits the number of disk areas requested to 25. The file type in the original FPB is then changed to indicate that the file was directed to disk or tape intermediate storage.

BACKUP FILE CONTROL INFORMATION

The first block in any backup file is filled almost entirely with control information. This information is used by SYSTEM/BACKUP when the file is printed or punched. The first twenty-four bits of the block will contain the logical record size, in bits, as computed by the prior portion of the OPEN routine. The next six bits of the block wil contain the number of bits that the record size was incremented to make it modulo forty-eight, if the backup medium was magnetic tape. If the backup medium was disk, these six bits will be equal to zero. The next eighteen bits specify the control information size, in bits. This field will contain the number of bits which are used in the first block of the file to contain the control information, exclusive of the File Parameter Block and the label. In the 9.0 version of the MCP, this number will be equal to 100, although all of the 100 bits may not be used.

The next twenty-four bits of the block will specify the FPB size, in bits. This number may vary from release to release. For the 9.0 version of the software, the FPB size is 1668 bits. The next twenty-four bits will contain the size of the label, if any, associated with the printer or punch file. This field will always contain these values, regardless of whether the file is

labeled or not. The next four bits will contain a number which specifies the type of label that is contained in the label area. In all cases, at the present time, this number will be either zero, indicating a standard label, or one, indicating that the file is unlabelled.

Unless the computed logical record size of the file is exactly equal to the size of the control information listed above, 100 bits for the 8.0 version of the MCP, a filler will be added after the control information. This filler will be of a size sufficient to make the next field in the first block, the FPB, begin on a logical record boundary. For example, if the original logical record size was 132 bytes and the backup medium was disk, the filler would consist of 964 bits.

The next field in the first block of the file will be the original File Parameter Block as it appeared in the user program and before any changes were made by the OPEN routine. Only pertinent information, delimited by the size specified by FPB.SIZE will be included. Following the FPB, another filler will probably be required to make the next field in the first block, the original file label, begin on a logical record boundary.

Actually, sixteen bits of spacing information precedes the file label; the spacing information thus begins on the logical record boundary. For the label, all of the sixteen bits will be set to zero. These sixteen bits will be followed by the label, which is constructed exactly as if the file had been directed to its intended medium originally. The label is always constructed and stored in the Backup file, regardless of whether the original file was labelled or not. SYSTEM/BACKUP may or may not cause the label to be printed or punched, depending upon whether the file was or was not labelled. The label in the first block will be followed by a filler, if necessary, to allow the first logical record of output data to begin on a logical record boundary within the block. The first block will always contain at least one logical output record.

BACKUP FILE LOGICAL RECORD FORMAT

Each logical record in the file will consist of sixteen bits of formatting information followed by the user's output data, unaltered. If the logical record was generated by a Position communicate operator, the contents of the data field are undefined and are ignored by SYSTEM/BACKUP. The sixteen bits are defined as follows.

Beginning with the 9.0 version of the software, the sixteen bits of carriage control information are subdivided as:

```
01 CARRIAGE_CONTROL                    BIT (16),
   02 FILLER                           BIT (3),
   02 BEFORE_AFTER                     BIT (1),
   02 CHANNEL_OR_SPACING               BIT (8),
   02 TYPE                             BIT (4);
```

In the description above, the BEFORE_AFTER field is applicable on WRITE operations which are directed to a printer file. A one in this bit position indicates the operation was WRITE AFTER ADVANCING. The CHANNEL_OR_SPACING field corresponds to the eight bits of spacing information passed on a WRITE communicate in the CT.ADVERB field in the communicate operator. These bits are defined in the Demand Management section of this document, but the definition is repeated here for reference.

```
CHANNEL_OR_SPACING
     = 0000 - No paper motion
     = 0001 - Skip to Channel One
     = 0002 - Skip to Channel Two
        .
        .
     = 1011 - Skip to Channel Eleven
     = 1100 - Skip to Channel Twelve
     = 1101 - Skip to first line of the form (1500 LPM
              printer only)
     = 1110 - Single space
     = 1111 - Double space
```

The TYPE field in the description provides information on the type of communicate issued by the user on this record. The CARRIAGE_OR_SPACING value will have different meanings, depending upon the value of the TYPE field. The correspondense between the two is shown below.

TYPE	Operation	CARRIAGE_OR_SPACING Value
0000	WRITE	Printer Channel Number
0001	WRITE	Punch Stacker Number
0010	SPACE	Number of Records to Position
0011	SPACE	Printer Channel Number on Position
0100	WRITE	Printer Spacing Information

Relative Files

A Relative file consists of records which are identified by relative record numbers. The file may be thought of as composed of a serial string of areas, each capable of holding a logical record. Each of these areas is denominated by a relative record number. For example, the tenth record is the one addressed by the relative record number 10 and is in the tenth record area, whether or not records have been written in the first through the ninth record area. Relative files are implemented using direct files.

Direct Files

Direct is the primitive file organization. A direct file is divided into a number of "record slots" of fixed length, each of which may contain one record. A record slot is "empty" if it contains no valid record. Full record slots may be made empty by deleting the record they contain, making the contents unaccessable through the normal mechanism. Since all bit patterns are potentially meaningful as data, a separate area in each block of the file is maintained to indicate which record slots within that block have been used. There will be one such "Presence Bit" for each record slot in that block and the bit vector thus formed is known as the Block Control Information (BCI). The user is not allowed to have access to the Block Control Information under normal circumstances.

Relative File Data Structure

The Relative file is a direct file. The blocks of the Relative file contain Block Control Information (BCI) as well as data records. The number of data records in a block is conatined in the "Records per Block" field of the disk file header in the case of an existing file. Originally, of course, this number is specified by the user programmer in his File Declaration. The data records will be located on byte boundaries to conform with the addressing capabilities of the B1000 Interpreters. The BCI will therefore be padded with zeroes to insure this. When a Relative file is originally created, all of the record slots are empty. Consequently, the presence bits in the BCI must be initialized when the area is allocated.

Relative File Disk Initialization

The use of presence bits to indicate that a record has been written into an available record slot means that disk areas that are allocated to a Relative file must be initialized when they are allocated. All presence bits in the Block Control Information must be set to zero at this time.

When a disk area is required, the MCP will be responsible for allocating the area, and will also be responsible for initializing presence bits. If the access mode of the file is sequential, the MCP just allocates the area and the Logical I/O routines will initialize each block before accessing it. If the access mode is random or dynamic, the MCP will initialize the entire area being allocated by automatically executing a special initialization program which will run at the user's priority. The user will have the option of executing this program himself, prior to executing the program which accesses the file, to initialize the entire file or any areas he choses. In the sequential mode, if the file is closed with the EOF pointer not at the end of an area, the MCP will initialize the remainder of that area.

The program which initializes newly allocated disk areas for Relative files is called SYSTEM/REL.INIT. If this program is called automatically by the MCP as described above, the program which requested the new disk area will not be allowed to execute until SYSTEM/REL.INIT has completed the initialization of the new area.

Relative File Parameter Blocks (FPBs)

The FPB for a relative file is the same as for a Conventional random file except that FPB.ACCESS is set to a value of 2, indicating Relative organization.

Relative Disk File Headers (DFHs)

The DFH for a relative file is the same as for a Conventional file except that the block size field will include the size of the block control information.

Relative File Information Blocks (FIBs)

The FIB for a relative file is the same as for a Conventional random file, except that a field which identifies the file as being Relative has been added. The field is named the FIB.ORGANIZATION field and can assume values of zero, indicating a Conventional or ANSI '74 Sequential file, one, indicating a Relative file, and two, indicating an Indexed Sequential file.

Buffers for Relative files will be the same as for Conventional files. They will be allocated when the file is opened with one I/O descriptor for each buffer and the buffer size equal to the block size, which is equal to the record size times the number of records per block plus the size of the block control information (1 bit/record made modulo eight).

Buffer management for Relative files will depend on the user's access method - Sequential, Random or Dynamic. For Random access the management of the buffers will be the same as that for Conventional random files. READ operations will be initiated on demand and WRITE operations will be initiated immediately after the logical I/O operation has occurred. If the access mode is Sequential, the buffer management will be the same as that for Conventional serial files. The Open procedure will fill all of the buffers and the Operating System will try to stay ahead of the user program, initiating physical Read operations when the last logical record in a buffer has been delivered to a user and initiating physical Write operations when the last logical record of the buffer is received.

The Dynamic access mode in ANSI '74 COBOL allows the user to switch between the Random and Sequential modes. In the Dynamic access mode, when switching from Sequential to Random, the last block is written to disk if it has been updated. When switching from Random to Sequential, the SMCP is called on to fill the buffers as if an OPEN or Position had occurred. In the Dynamic access mode, the access mode desired, Random or Sequential, must be specified in the communicate operator generated by each logical READ operation.

Relative File Communicate Operators

Three new communicate operations, corresponding to the verbs DELETE, START and REWRITE have been added to the 9.0 Operating System. To simplify the implementation and to avoid potential file equivalence problems, new communicate operations for relative files have been added to the software, rather than modifying an existing operation. The READ, WRITE and REWRITE

communicate operators have a format which is similar to the format for the READ, WRITE and REWRITE communicate formats for conventional files. The format for the DELETE operation, on Relative files, is similar to the format for the same operation on Indexed Sequential files. The ANSI '74 COBOL START verb has been implemented as a new communicate and is handled by the Micro MCP.

Indexed Sequential Files

Indexed Sequential files consist of two new primitive file types: Direct files and Index files. For each Indexed Sequential file there is one and only one data file and this file is implemented as a Direct file. For each key of the Indexed Sequential file there is a corresponding file of type "index". In the MCP code, these two types are listed as INDEX.SEQ.DATA.SET.FILE and INDEX.SEQ.INDEX.FILE; they will be refferred to as Direct files and Index files in this document.

Direct Files

Direct files were discussed in the documentation on Relative files. A portion of that discussion is repeated here for convenience. More details will be found in the preceeding discussion. A Direct file is a primitive file type that is divided into a number of "record slots" of fixed length, each of which may contain one record. A record slot is "empty" if it contains no valid record. Full record slots may be made empty by deleting the records they contain, making the contents of that slot inaccessable by the normal mechanism. Since all bit patterns are potentially meaningful as a record, a bit flag is maintained for each record slot to show the validity of its contents.

Since all record slots are the same size (MAXRECSIZE) the absolute disk address can be easily calculated from the record slot number. The file is divided into groups of record slots called "blocks", each consisting of "blocking factor" record slots plus the "Block Control Information", a bit mask which indicates the presence of a valid record plus enough filler bits to make the container modulo eight. There is a significant difference between the Block Control Information for a Direct file and an Index file, however.

Index Files

An Index file is the second new file type. Index files contain fixed length records organized in tables with Block Control Information to describe the table. Each block of an Index file will constitue a separate table. The importance of this fact will be explained later.

The records in the Index file consist of Key/Address pairs. The addresses point to other tables in the Index file or to records of the Index Sequential file's data file, the Direct file. The tables in the Index file form a tree structure and the records in the table are ordered by Key value to allow fast random access. The tables whose entries point to data records are linked together to allow fast sequential access.

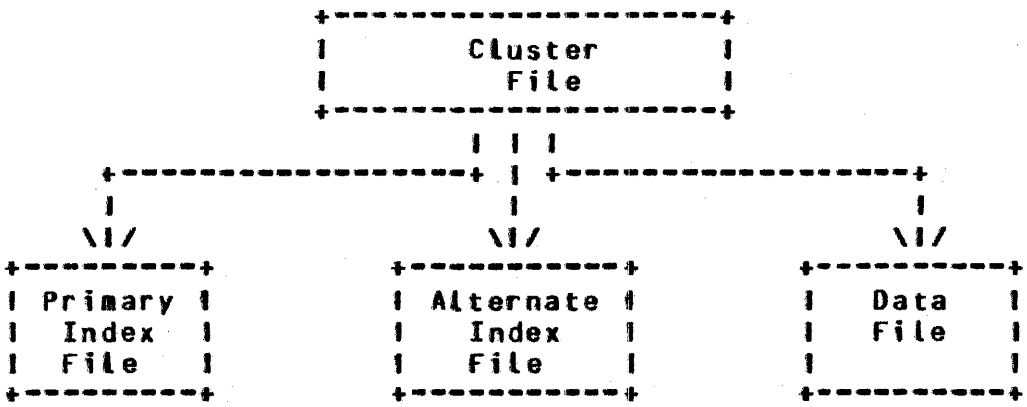
Cluster Files

In addition to these two new file types, there must reside, somewhere on disk, information relating all of the various files which compose an Indexed Sequential file. This information is maintained, by the MCP, in a third new structure which will be a separate conventional file on disk and which will be known as a "Cluster" file. The name of the Cluster file will correspond to the user's declared name for his Indexed Sequential file. In the MCP code, this file type is referred to as an INDEX.SEQ.GLOBAL.FILE, though it will be called merely a Cluster file in this document.

The Cluster file provides the ability to reference the entire Indexed Sequential file structure by simply referencing the Cluster file. When the Compilers generate code which applies to Indexed Sequential files, they actually reference the Cluster file. The Cluster file will contain the names of the other files associated with the Indexed Sequential file. As mentioned previously, there will be one Index file for each key listed in the Indexed Sequential file.

The statement above does not mean that all of the Index files will be opened when a Cluster file is opened. The Index files are only opened when they are first referenced in the program and this actually happens automatically. The compilers do not generate code to open the Index files. The MCP simply detects that the referenced Index file has not yet been opened, obtains the necessary information from the Cluster file, and opens the file.

The Cluster file does require an additional Disk File Header in memory, but only while an Indexed Sequential file is being opened. It is not necessary for it to be in memory after the file has been opened. The Cluster file also adds an entry to the user's disk directory. The diagram below shows a Cluster file schematically. This particular file has one primary, or "Prime" Key and one Alternate Key.

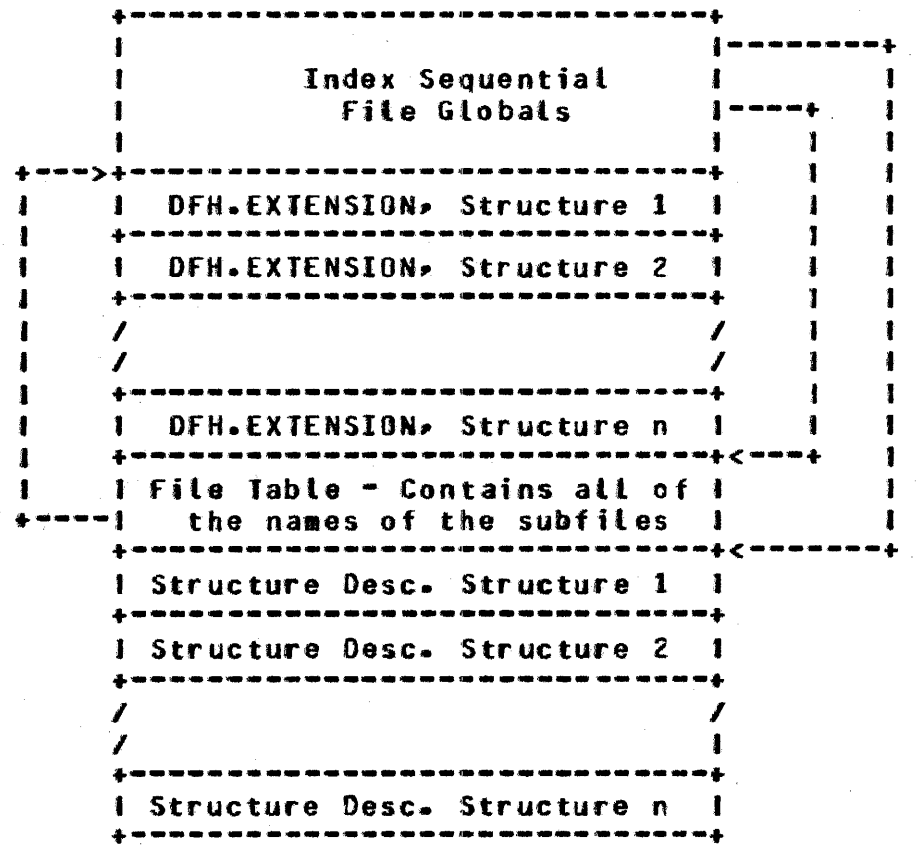


This organization for Indexed Sequential files offers several advantages over any other. Each file, the file which contains the actual data and all of the Index files, will have fixed record and block sizes. This will simplify the problem of managing the buffers that are assigned to the files. Both of these file types are nothing more than Conventional files with some order imposed upon the contents of the file. Consequently, the Disk File Headers, or "File Descriptors" required for each file are the same as those for Conventional files. This is discussed in more detail later in the document.

Conceptually, this mechanism is easier to visualize and implement than would be multiple structures residing in one physical file. Also, any of the files may be located on different spindles, which will clearly improve performance, since arm movement time may be overlapped, and access to all of the files may occur asynchronously. The Direct file and the Index file may be accessed independently of each other.

The design does impose certain restrictions, which fall in the category of "operational" restrictions and which do not impact performance. A checking mechanism is required to insure the integrity of files which are accessed independently. The MCP must insure that the correct version of the Index file is used with its corresponding Direct file. Also, some extra memory for Disk File Headers will be required, since more actual Headers will be required. A naming convention for all of the files must be imposed, thus removing some small amount of generality from the user's capabilities. This may actually be an advantage, however. The naming convention is implemented in the Compiler, not in the MCP, though this may not be apparent to, and should not be important to the user.

The Cluster file is a Conventional data file which contains the information relating all the component files of the Indexed Sequential file. The structure of the Cluster file is similar to the Data Base Dictionary format in the Data Management System.



The DFH.EXTENSION and Structure Descriptor fields shown above are both discussed in the paragraphs that follow. The pointer shown above from the File Table is one of many. There is an entry for each file in the File Table and each entry has a pointer to its associated DFH.EXTENSION.

Indexed Sequential Data File Structure

The data file of an Indexed Sequential file is a Direct file. The blocks of the data file contain Block Control Information (BCI) and data records, similar to the blocks of a Relative file as presented previously. The number of data records in a block is specified by the Records per Block field of the disk file header. A similar structure is used on Indexed Sequential files in the Data Management System. Block Control Information for the Index files associated with all Indexed Sequential files is significantly different from that for Relative files.

Indexed Sequential Index File Structure

Index files contain records consisting of Key/Address pairs within a block. The file itself is a tree structure whose nodes are blocks. Each block of the file is a node or table. The first node is the root table. The root table and tables on all levels except the last are called coarse tables. The tables in the last level of the tree are called fine tables. Entries in coarse tables point to the next level table whose highest entry matches the key of the coarse table entry. Fine table entries point to a record in the Direct file whose key matches the fine table entry (See Figure 3). Fine tables are linked together in logical order to provide fast sequential access and easier Current Record Pointer (CURRENT) maintenance.

The addresses in these tables are not absolute disk addresses. Instead, they are thirty-two bit combinations of an area number, a segment number within the area and a displacement into the segment. This displacement is merely the record number within the block. All addressing of Index tables as well as of records in the data file is accomplished on a relative basis as opposed to an absolute one.

The blocks in Index files contain Block Control Information of a different content and format. The format and content of the Block Control Information maintained in an Index file is shown below. A similar structure exists for DMS Index files.

```
01 INDEX.FILE BCI                          BIT (38),

  02 BC.TYPE                               BIT  (2),% 0=COARSE, 1=FINE
  02 BC.PRESENT.RECORD.COUNT               BIT (12),
  02 BC.NEXT.LOGICAL.BLOCK                 BIT (24);% VALID FOR FINE TABLES
```

The individual records in the Index files have a fixed format; since the Key specified by the user must be contained in these records, the size of the records may vary with the keys but the format will always be as shown below. The same format is used by the Data Management System for records in Index tables.

```
01 INDEX.RECORD DECLARATION,% FOR DMS AND ANSI '74 INDEX FILES

  02 IR.POINTER,
    03 IR.AREA.DISP,
      04 IR.AREA.NMBR                BIT(8),
      04 IR.SEG.NMBR                 BIT(16),
    03 IR.OFFSET                     BIT(8), % VALID FOR FINE TABLES
  02 IR.KEY                          BIT(KEY.SIZE);
```

The organization of an Index file is shown in the diagram below.

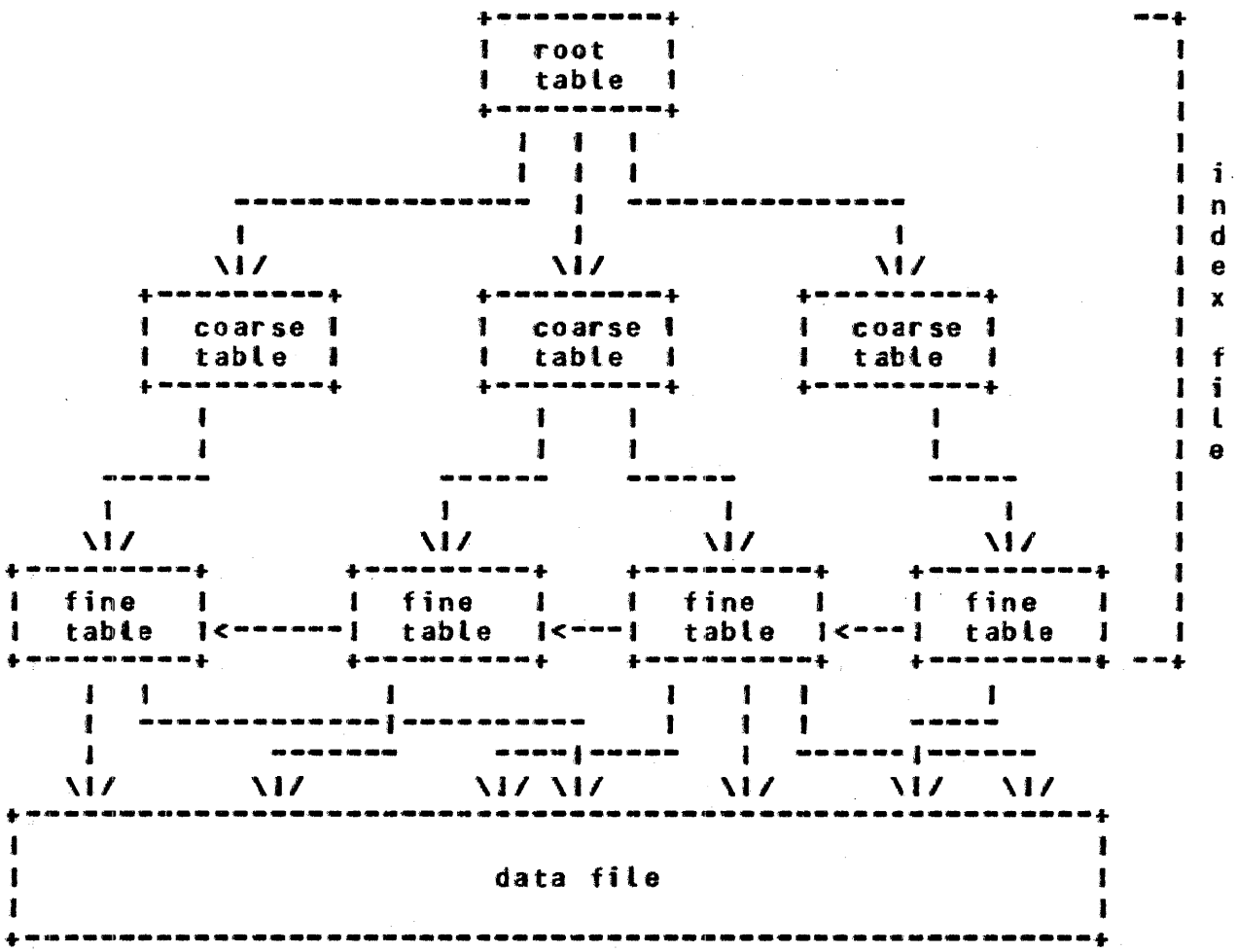


Figure 2 - Index File Organization

This structure for the Index files allows the implementation of the most efficient search and addition algorithms. Linking the last level of the fine tables together allows efficient sequential access of the records in the data file. Using this link, the CURRENT need only point to the last entry accessed in the fine tables and not to the path through the coarse tables to the fine tables. This eliminates the need for restrictions on the number of levels allowed in order to maintain the CURRENT.

It also makes checking for changes in the CURRENT, caused by other users accessing the file, easier.

The File Parameter Block (FPB) of the Cluster file of an Indexed Sequential file will be positioned in the code file among the other FPBs according to the order of it's declaration in the user's source code. In addition to the information normally contained in an FPB for a Conventional file, a Cluster file FPB will contain a type field which identifies it as a Cluster file FPB, a pointer to the data file FPB and an integer which indicates the number of keys associated with the Index Sequential file. There will be one FPB for each Key declared and these FPBs will immediately follow the FPB for the data file in the code file of the program. This is shown in the diagram in Figure 3.

Default values are used for the file attributes of a Cluster file. The user may not change these values. The number of disk areas will be set to one, records per block will be set to one, block size will be set to 180 bytes and blocks per area will be set to 50. The ALL.AT.OPEN boolean will be set, causing the disk area to be allocated when the file is opened for the first time.

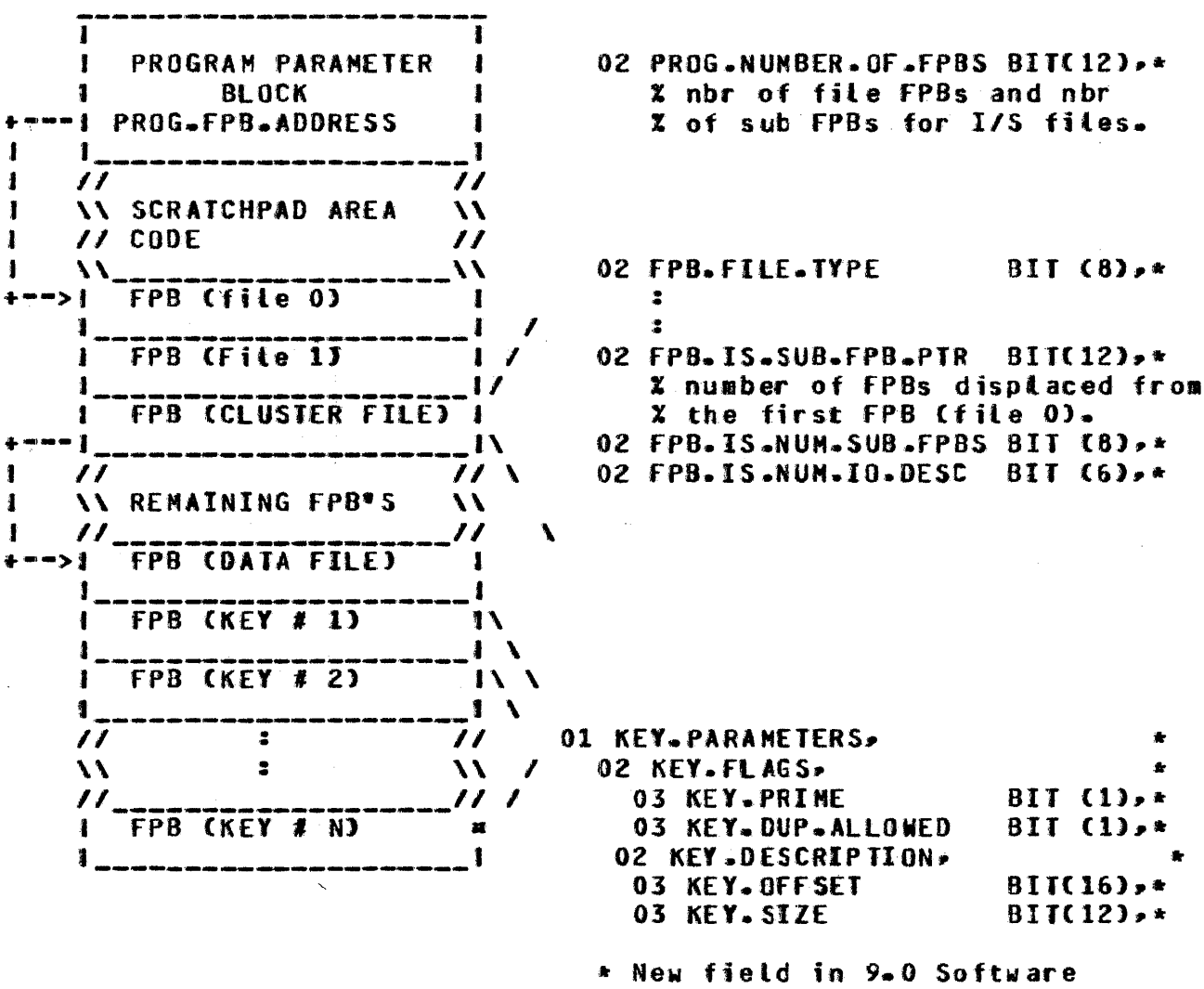


Figure 3 - Code File on Disk

Some changes were also necessary in the Program Parameter Block in the 9.0 software. The changes are required to prevent programs which contain Relative and Indexed Sequential files from being executed on versions of the MCPs released prior to the 9.0 version. Further, program code files which are executed under control of the 9.0 MCP may no longer be executed under control of any prior MCPs. For this reason, users who anticipate returning to prior versions of the MCP are advised to retain copies of their code files and to not execute these copies under control of the 9.0 software.

Indexed Sequential Memory Structures

Generally, the memory structures used in the Indexed Sequential implementation are much like the current Data Management System memory structures, with but a few exceptions which take advantage of the more specific requirements of the ANSI '74 COBOL definition. Unlike DMS, which does not use File Information Blocks in memory, Indexed Sequential files will have an FIB dictionary entry which will point to an Indexed Sequential FIB. Since the files may be shared among the programs that are executing, this FIB will contain only the information pertinent to a specific user and will be referred to as the User Specific Information (USI) field.

The USI will contain a pointer to the file specific information, the information that relates only to the file itself regardless of who is using it. The central element in this structure is the information necessary to relate the various component files of the Indexed Sequential file. This is actually global information, global to all of the users, and will contain a table whose entries point to information specifically concerning the component file. The structure which contains this information is referred to as the Index File Structure Descriptor (STR). There will be one Structure Descriptor for the data file and one for each Index file associated with the Indexed Sequential file.

Structure Descriptors contain pointers to the DFH, Buffers and CURRENT information associated with the various Index files. The relationship of the various memory structures used is shown diagramatically in Figure 4.

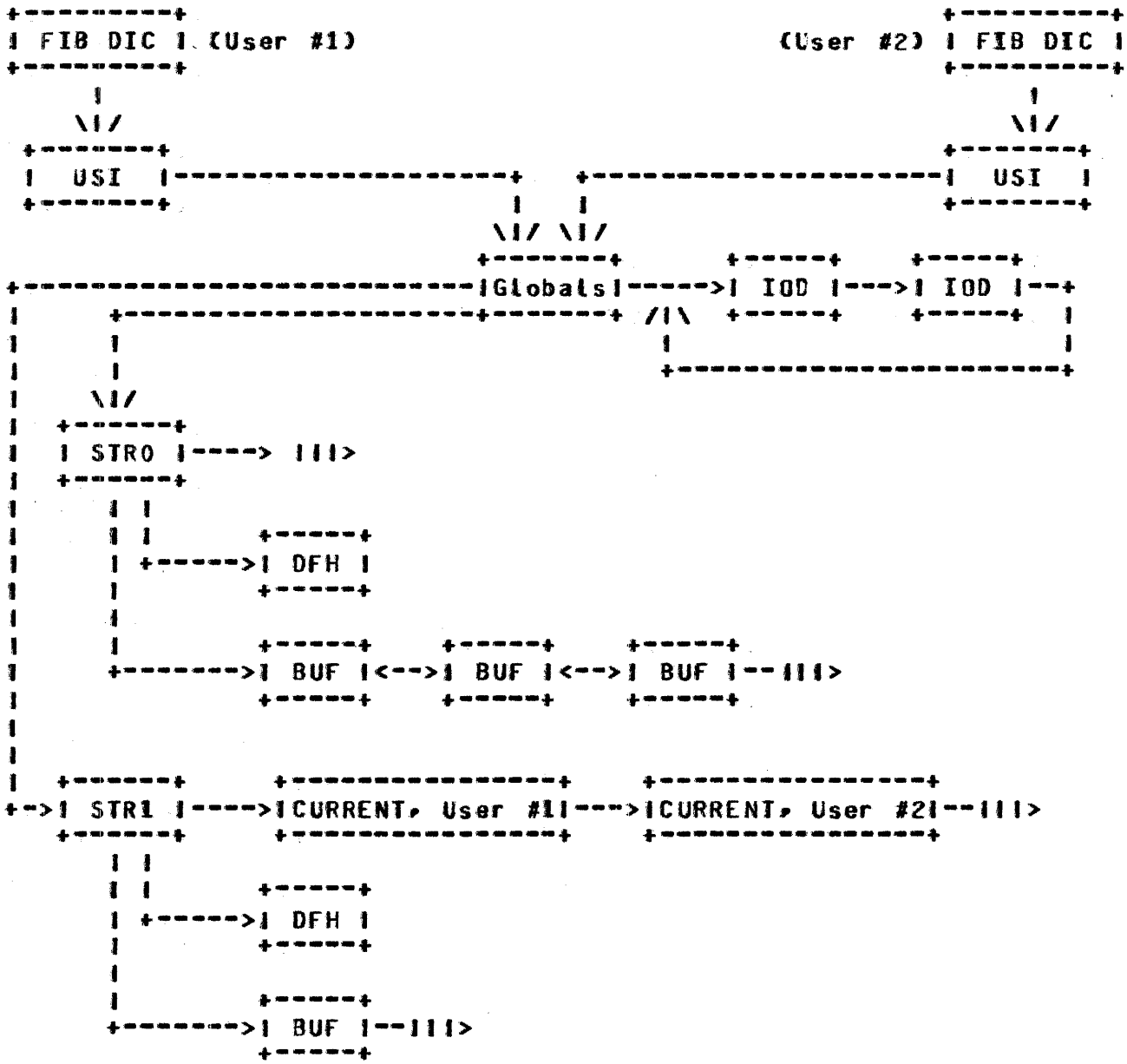


Figure 4 - I-S File Memory Structures

EIB Dictionaries

From the user's view point, Indexed Sequential files are more like a Conventional Random file, except for the fact that symbolic key values are used, than they are like DMS structures. Though the Data Management System is a superset of the Indexed Sequential implementation, the user is more likely to have several small and transient Indexed Sequential files than one large file which he would treat as a data base.

A secondary, but important, goal of the design of the ANSI '74 COBOL implementation was to allow a smooth integration of Relative and Indexed Sequential files with the Conventional file mechanism. For this reason and for other reasons, access to an Indexed Sequential FIB is via the FIB Dictionary, which is also used to access Conventional file FIBs. The FIB for an Indexed Sequential file is itself quite different from the FIB for a Conventional file. The Indexed Sequential file is associated with several physical files, whereas the Conventional file is associated with only one. Also, more than one user may share the information, including the data buffers, of an Indexed Sequential FIB; a Conventional file FIB is used by only one user. If two users are accessing the same physical Conventional file, each user will have his own FIB.

For these reasons, an Indexed Sequential FIB contains three major parts:

1. User Specific Information
2. File Global Information
3. Component File Specific Information

The entry in the FIB dictionary corresponding to the Indexed Sequential file points to the User Specific Information (USI) of this Indexed Sequential FIB.

Indexed Sequential User Specific Information (USI)

The USI contains information associated with one user only. The MCP must know how the user has opened the file, for example as INPUT, and how the user is accessing the file, such as sequentially. This information is kept in the USI. User statistics, status and MCP workspace are also kept in this structure. Finally, there is a pointer to the next part of the Indexed Sequential FIB, the global information associated with the physical file.

```
01 USER.SPECIFIC.INFORMATION,

   02 FIB.COMMON.PORTION                    BIT(220),% The first
      03 FIB.BOOLEANS                       BIT(58), % 220 bits of
         04 FIB.OPEN                        BIT(1),  % USI are the
         04 FIB.CLOSING                     BIT(1),  % same as
         04 FIB.OUTPUT                      BIT(1),  % Conventional
         04 FIB.INPUT                       BIT(1),  % FIBs
      03 FIB.ORGANIZATION                   BIT(4),  %
         % 1 = RELATIVE
         % 2 = INDEXED/SEQUENTIAL
   02 USI.FIB,
      03 FIB.USI.NOT.FIRST.TIME.THRU             BIT(1),
      03 FIB.USI.LAST.OP.READ                    BIT(1),
      03 FIB.USI.DUPLICATE                       BIT(1),
      03 FIB.USI.MATCH.FOUND                     BIT(1),
      03 FIB.USI.UPDATE.FLAG                     BIT(1),
      03 FIB.USI.FIRST.PASS                      BIT(1),
      03 FILLER                                  BIT(2),
      03 FIB.USI.ACCESS.MODE                     BIT(4),
      03 FIB.USI.JOB.NUMBER                      BIT(24),
      03 FIB.USI.RECORD.ADDRESS                  BIT(24),
      03 FIB.USI.KEY.POINTER                     BIT(24),
      03 FIB.USI.COMMUNICATE.WORKSPACE,          BIT(616),
         04 FIB.USI.BINARY.SEARCH.ARGUEMENTS     BIT(208),
         04 FIB.USI.INTERFACE.PADS               BIT(96),
         04 FIB.USI.SAVE.STATE.AREA              BIT(312),
      03 FIB.USI.GLOBAL.POINTER                  BIT(24),
      03 FIB.USI.CURRENT.STRUCTURE               BIT(8),
      03 FIB.USI.HEADER                          BIT(24);
```

Indexed Sequential File Global Information (GLOBALS)

As shown in the above diagram, the first 220 bits of the User Specific Information are the same as the first 220 bits of an FIB for a Conventional file. The rest of the information can be seen to be items that are peculiar to a specific user of the structure. It is information that is necessary for Operating System storage of the "state" variables that may be required to perform a single operation for this user.

Included in this information is a pointer to the next portion of an Indexed Sequential FIB, the file Global information. This information, known as the GLOBALS field, contains information about the various physical files which comprise an Indexed Sequential file. Its main function is to provide a path to the required files necessary to complete an I/O operation. A secondary function is to store information global to the Indexed Sequential file.

The path to a particular component file is provided by a system descriptor contained in a table of system descriptors. The first entry of the table points to the data file. The remaining entries point to Index files, one for each key declared; they appear in the order of the declaration of their corresponding keys. For any operation which specifies a key, the compiler will specify the key number, which will be used as an index into this table.

The global information consists of pointers to the chain of I/O descriptors to be used for operations on the Indexed Sequential data file, a count of users who are updating the file, and Lock bits to support ANSI '74 COBOL's file level lockout. Also contained in GLOBALS are the count and flag fields necessary to enforce the prohibition on concurrent updates. A programmatic description is shown below.

```
01 GLOBALS

   02 GLOB.VERSION.NUMBER              BIT(8),
   02 GLOB.NUMBER.OF.USERS             BIT(6),
   02 GLOB.NUMBER.OF.UPDATERS          BIT(6),
   02 GLOB.DISK.COPY.ADDRESS           DSK.ADR,
   02 GLOB.SIZE.IN.BITS                BIT(16),
   02 GLOB.MEMORY.ADDRESS              BIT(24),
   02 GLOB.LOCK.BITS                   BIT(2),
   02 GLOB.IO.DESC.CHAIN.ADDRESS       BIT(24),
   02 GLOB.MAX.STRUCTURE.NUMBER        BIT(8),
   02 GLOB.FLAGS                       BIT(6),
      03 GLOB.DMS.FILE                 BIT(1),
      03 FILLER                        BIT(4),
      03 GLOB.WRITE.ERROR              BIT(1),
   02 GLOB.CONCURRENT.INFO,                    % AT THIS DMS INFO AND
      03 GLOB.INUSE.COUNT              BIT(6), % IS INFO ARE DIFFERENT
      03 GLOB.CONCURRENT.FLAGS,                % TIL STR DIRECTORY
         04 GLOB.FILE.AVAIL            BIT(1),
         04 GLOB.UPDATE.REQUIRED.OR.INPROC BIT(1),
   02 GLOB.STRUCTURE.DIRECTORY,
      03 GLOB.STRUCTURE.DESCRIPTOR     SY.DESC,
```

All of the pointers to subsequent portions of the Indexed Sequential structure, all of which are known as Structure Descriptors, are contained in the GLOBALS field. This simplifies the task of maintaining the structures and it allows the buffers to be shared among the various users. It adds one level of indirection to all accesses to the data of course, but this expense is small for the benefits it yields.

The Structure Descriptor is similar to an FIB for a Conventional file except that all of the User Specific Information is removed and maintained in the USI field. For the Index files of an Indexed Sequential structure, necessary key information is also kept in the Structure Descriptor. For example, the position of the key within the data records, it's size, whether or not duplicates are allowed, and whether or not it is the prime key are all stored in the STR. A programmatic description is shown below.

```
01 STRUCTURE_DESCRIPTOR,

   02 STR.NUMBER                          BIT(8),
   02 STR.TYPE                            BIT(4),
   02 STR.USER.COUNT                      BIT(6),
   02 STR.BUFFER.LOCK                     BIT(2),
   02 STR.BUFFER.LIST.POINTER             BIT(24),
   02 STR.RECORDS.PER.BLOCK               BIT(8),
   02 STR.SEGMENTS.PER.BLOCK              BIT(8),
   02 STR.RECORD.SIZE                     BIT(16),
   02 STR.BLOCK.SIZE                      BIT(16),
   02 STR.BLOCKS.PER.AREA                 BIT(16),
   02 STR.SEGS.PER.AREA                   BIT(16),
   02 STR.DFH.ADDRESS                     BIT(24),
   02 STR.DFH.OFFSET.TO.EXTENSION         BIT(16),
   02 STR.CURRENT.POINTER                 BIT(24),
   02 STR.FLAGS
      03 STR.PRIME.KEY                    BIT(1),
      03 STR.DUPLICATES.ALLOWED           BIT(1),
      03 STR.SIMPLE.KEY                   BIT(1),
   02 STR.SPLITFACTOR                     BIT(12),
   02 STR.KEY.INFO,
      03 STR.NUMBER.OF.KEYS               BIT(8),
      03 STR.SUB.KEY,
         04 STR.ITEM.OFFSET               BIT(16),
         04 STR.ITEM.SIZE                 BIT(12);
```

As shown in Figure 4, the Structure Descriptor contains a pointer to the Disk File Header, the MCP-defined structure which is at the last level. This structure, as it always has, contains information relating almost exclusively to the physical characteristics of the file. Any logical information in the header, such as record size and records per block, was obtained from the program which originally created the file.

The format of the disk file header had to be expanded in the 9.0 version of the software to accomodate the ANSI '74 COBOL implementation. Prior to the creation of the 9.0 version, several pieces of information associated with DMS Data Bases, which should have been part of the DFH, were maintained

separately due to a lack of available space in the then curent definition of the disk file header. These fields have also been incorporated in the new disk file header. The new format has been designed to prevent the occurrence of such problems in the future, whenever the need for new fields in the DFH arises.

Disk File Header Extensions

Some efficient means of available disk space maintenance had to be devised for Indexed Sequential files. To accomplish this, the necessary information regarding the available space is maintained in the Cluster file as a data record. When an Indexed Sequential file is opened, this information is brought into memory and stored in a memory area which will immediately follow the Disk File Header for the data file. This area is known as the Disk File Header Extension.

Indexed Sequential Disk File Header Extension

When the Indexed Sequential file is opened, the information on the available space within the Direct file, all of which space is not available as far as the system is concerned, is brought into memory and stored in the DFH Extension. The format of this information in memory is as shown below.

```
01 DFH.IS.EXTENSION,

  02 FILLER                          BIT(16),
  02 DFH.IS.EXTENSION.SIZE           BIT(16),
  02 DFH.IS.EXTENSION.VERSION        BIT(36),
  02 DFH.IS.NEXT.FREE.RECORD         BIT(32),
  02 DFH.IS.NEXT.FREE.BLOCK          BIT(32),
  02 DFH.IS.ROOT.TABLE               BIT(24),
  02 DFH.IS.UPDATE.FLAG              BIT (1);
```

Indexed Sequential Available Space Allocation

The Indexed Sequential file system maintains two fields in the DFH.EXTENSION of each file which keep track of available space within the Direct file. This available space should not be confused with the available disk space that is maintained by the system. Available space in an Indexed Sequential file or in a Relative file means that a record has never been written into an available record slot or that a record was written at some time but was subsequently and is now deleted. To the system, all of the space allocated to the file is in use and none of it is available.

Both of the available space pointers shown above, DFH.IS.NEXT.FREE.RECORD and DFH.IS.NEXT.FREE.BLOCK, will contain addresses of blocks which have available space. The NEXT.FREE.RECORD pointer does not actually point to a record but points to the block which contains the available record slot. Record slot allocation within a block is accomplished using the presence bits in the Block Control Information for that block.

The DFH.IS.NEXT.FREE.BLOCK field will contain the area and block number of the next totally available block at the logical end of the file. The first disk area of the data file is allocated when the file is first opened and the NEXT.FREE.BLOCK field is set to zero, a valid address, at that time. Also, when the file is first opened, the NEXT.FREE.RECORD field is set to @FFFFFFFF@. When the Micro MCP needs to add a record to the file and the NEXT.FREE.RECORD field contains @FFFFFFFF@, it means that no records are available in a block that has already been initialized. The allocation must be accomplished using the NEXT.FREE.BLOCK field.

The Micro MCP will then initialize the Presence Bits in the Block Control Information of the block addressed by the NEXT.FREE.BLOCK field, move the address which is in the NEXT.FREE.RECORD field, in this case @FFFFFFFF@ to the first thirty-two bits of the last record slot in the block, move the address of this block to the NEXT.FREE.RECORD field and increment the NEXT.FREE.BLOCK field. If the incremented value of the NEXT.FREE.BLOCK field causes this disk area to exceed the specified size of a disk area, @FFFFFFFF@ will be stored in the NEXT.FREE.BLOCK field instead. The use of this value is discused in a subsequent paragraph.

The record which is being added is then moved to the first record slot in the newly allocated block, the presence bit for this slot is set and the block is written. The presence bits for the second and all subsequent record slots within that block will be set to zero, due to the initiaiization process. @FFFFFFFF@, the value that was previously in the NEXT.FREE.RECORD field, will be stored in the first thirty-two bits of the last record slot in the block.

When the next record is added to the file, the Micro MCP will again examine the NEXT.FREE.RECORD field and it will now contain the address of the block that was just allocated. The Micro MCP will read the block into memory, if necessary, and examine the Presence Bits in the Block Control Information. The first available record slot will be the second slot within the block. The Presence Bit for this slot will be set and, if this is the last record slot in the block, the @FFFFFFFF@ stored in the first

thirty-two bits of the record slot will be moved back to the NEXT.FREE.RECORD field, and the record will then be stored in the slot. If the second record slot is not the last in the block, @FFFFFFFF@ will remain in the actual last slot and the NEXT.FREE.RECORD field will not be changed.

Allocation in the Direct file will proceed in this manner, asuming that no DELETE operations are performed, until the disk area becomes filled and, as mentioned previously, @FFFFFFFF@ is stored in the NEXT.FREE.BLOCK field. This value serves as an indicator to the Micro MCP that the next disk area has not yet been allocated by the S.MCP. When the Micro MCP encounters this value, it merely passes control to the S.MCP which will allocate the area and store its address in the disk file header and in the NEXT.FREE.BLOCK field. The Micro MCP will then initialize the Block Control Information and proceed as was described previously.

The process just described may be interrupted by the occurrence of a DELETE request from a user. When this occurs, the address in the NEXT.FREE.RECORD slot is stored in the first thirty-two bits of the record being deleted, the Presence Bit associated with the deleted record is reset and the block is written to disk. The address of the block which contains the deleted record is then stored in the NEXT.FREE.RECORD field. The next time a record is added to the file, it will consequently be stored in the area occupied by the record that was just deleted and the NEXT.FREE.RECORD field will be restored to its prior value. This operation should eliminate the need to periodically rewrite the entire file to eliminate large numbers of empty record slots, a process commonly known as "garbage collection".

Should more than one record in a block be deleted, the Micro MCP only needs to insure that the first thirty-two bits of the last available record slot in that block contains the address of the next block in which a record slot is available or @FFFFFFFF@ if there is no such next block. This is true even if all of the records in a block are deleted. No pointers need be changed, in this latter case, until the next DELETE operation occurs. Assuming that no new records have been added in the interim, the Micro MCP then needs only to insure that the address of the block which is totally empty is stored in the slot previously occupied by the deleted record.

Allocation of space for an Index file associated with an Indexed Sequential file is somewhat simpler than for a Data file, since record availability does not have to be maintained. Whenever a record is deleted, the pointer to that record in the Index file is destroyed and the table contained in the block is compacted. The count of the actual number of entries in that block, which is

maintained in the Block Control Information of an Index file, is decremented. No other action is required for the Index file.

Maintenance of the NEXT.FREE.BLOCK field of an Index file is exactly like that for the data file. This field will always contain the address of the next available block at the logical end of the file. The Micro MCP will set the field to @FFFFFFFF@ when the next disk area must be allocated, exactly as is done for the data file.

The NEXT.FREE.RECORD field is used to address a linked list of blocks within the file that are completely empty. This can only occur when all of the records that were addressed through this block have been deleted, a situation which should seldom occur in actual use.

Index File Table Splitting

The "splitting" of fine tables in the Index file is an operation that is always performed by the S.MCP. Any time the addition of a record to the file causes a need for a fine table to be divided in two, the Micro MCP passes control to the SDL portion. Consequently, the S.MCP performs most of the available space maintenance for the Index files, while the Micro MCP performs the majority of this work for the data file.

Current Record Pointer (CURRENT)

The CURRENT is a structure that, for ANSI '74 COBOL, logically belongs in the User Specific Information field, since there is only one CURRENT per user. There are two reasons for associating the CURRENT with the Structure Descriptor, however. First, DMS has a CURRENT for each structure and a pointer exists in each STR to the appropriate CURRENT. To be compatible with DMS, each STR of an Indexed Sequential file points to the CURRENT for that structure. A current structure number is maintained in the USI to satisfy ANSI '74 requirements. Second, since the file can be shared, an operation by one user can affect the CURRENT of another user. To guard against this, each CURRENT is checked when an operation which can affect it is performed. To aid the search of CURRENTs, they are linked together, the first one being pointed to by the STR. A programmatic description of the CURRENT field is presented below.

```
01 CURRENT_DECLARATION,
   02 CUR.LINK                          BIT(24),
   02 CUR.JOB.INVOKE,
      03 CUR.CUR.JOB                    BIT(16),
      03 CUR.CUR.INVOKE                 BIT(6),
   02 CUR.STATUS                        BIT(2),  % 0-DEL, 1-VAL
   02 CUR.FINE.TABLE,
      03 CUR.AREA                       BIT(8),
      03 CUR.BLOCK                      BIT(16),
      03 CUR.RECORD                     BIT(12),
```

CURRENT Maintenance

The current is maintained for Indexed Sequential files which use either Sequential access or Dynamic access. When the user is accessing the file sequentially, the current is maintained for the key of reference (USI.CURRENT.STRUCTURE). For output files, the key of reference must be the prime key and CURRENT always points to the last entry written. For a new file, CURRENT is initialized to point to the first entry but CUR.STATUS is set to indicate the entry has not yet been written. For an old file opened OUTPUT EXTEND, the current is initialized to the last entry written. The Micro MCP uses the current on output files to insure that records are written in sequence, a requirement of ANSI 74 COBOL.

Sequential INPUT or INPUT-OUTPUT files require that the current points to the last record read. On the next READ operation, the current is incremented to point to the next available record. If the current record is deleted or the CURRENT was positioned by an OPEN or START, then CUR.STATUS is set to indicate that a record has not yet been read. The next READ will deliver the record and reset CUR.STATUS.

For files in Dynamic access mode, the meaning of CURRENT is more complicated. The CURRENT will be handled exactly as in the case of Sequential INPUT or INPUT-OUTPUT. This means that some sequences of operations may not produce the desired intuitive result. The example below illustrates the problem.

Consider the Index table at the right. What should the result of a READ NEXT be, in the following sequence of operations?

ABLE
DOG
GOLF

a. READ(ABLE), ADD(BAKER), READ NEXT;
b. READ(DOG), DELETE(DOG), ADD(ECHO), READ NEXT;
c. READ(DOG), DELETE(DOG), ADD(CHARLIE), READ NEXT;

For our implementation the READ NEXT produces the following results:
a. BAKER
b. GOLF
c. GOLF

Indexed Sequential Buffer Management

The method of allocating buffers in prior versions of the MCP and in the 9.0 version for Conventional files is known as Static allocation. This method of allocating buffers is simple, once the number of buffers has been chosen by the user. The buffers are merely allocated when the file is opened and they remain assigned to the file until it is closed. If the number of buffers allocated is too small, however, then operations upon the file may be inefficient. If the number of buffers allocated is too large, then nothing is gained in efficency and memory space is wasted.

On an Indexed Sequential file particularly, the number of buffers actually needed varies with the type of operation and the state of the Indexed Sequential file. The optimum number of buffers is best chosen dynamically to avoid the disadvantages mentioned above.

Allocating buffers on demand and deallocating them when the memory they occupy is required for other purposes is known as Dynamic allocation. Dynamic allocation has always been used for buffers associated with a DMS data base. It is accomplished by calling the MCP's memory allocation procedure, GETSPACE, whenever a buffer is required. Deallocation is accomplished by allowing GETSPACE to overlay DMS buffers when necessary. Dynamic allocation has also been implemented for Indexed Sequential files.

The management of buffers associated with an Indexed Sequential file presents a special problem for the MCP, since there can be a variable number of them, depending upon the operation, and they can be different sizes, depending upon which component file is being accessed. To solve the problems associated with a variable number of buffers, the Prioritized Memory Management algorithm,

developed for the 7.0 release should be used. This memory manager overlays buffers whenever space is needed and the priority of a buffer makes it a candidate to be overlaid. The FIFO Memory Management algorithm can be used but performance may be impacted on a multi-programming system.

To solve the problems associated with variable size buffers addressing the same Indexed Sequential file, all of the buffers used for one structure are linked together and pointed to by the structure, so that all buffers in a chain are of the same size.

Indexed Sequential Buffer Descriptor (BD)

The Buffer Descriptor is the structure used to maintain the buffers associated with the Indexed Sequential file. It contains the necessary link fields, identification fields, and state information. Since the memory manager may overlay the first buffer in a chain, the memory link field, ML.POINTER, will contain the structure address so that STR.BUFFER.LIST.POINTER may be updated. A programmatic description of the Buffer Descriptor is presented below.

```
01 BUFFER_DESCRIPTOR,
   02 BD.AREA.DISPLACEMENT,
      03 BD.AREA                        BIT(8),
      03 BD.OFFSET                      BIT(16),
   02 BD.USER.COUNT                     BIT(4),
   02 BD.IN.MEMORY                      BIT(1),
   02 BD.IO.ERROR                       BIT(1),
   02 BD.WRITER.CONTROL,
      03 BD.REQUIRES.A.WRITE            BIT(1),
      03 BD.CONTROL.POINT               BIT(1),
   02 BD.NEXT.BUFFER.DESCRIPTOR         BIT(24),
   02 BD.PRIOR.BUFFER.DESCRIPTOR        BIT(24);
```

I/O descriptors are shared among all the buffers. The BEGIN and END addresses in the descriptors may be modified when a descriptor is used by the Operating System. The number of buffers allocated depends on the number of active structures associated with the Indexed Sequential file. This technique serves to minimize the number of descriptors in the disk chain, thus reducing the amount of processing required by GISMO, and it minimizes the memory requirements for descriptors. It does require an allocation mechanism for descriptors, in addition to one for buffers, but this expense has been found to be worth the benefits.

Concurrent Update Operations

Concurrent READ operations on the same record of an Indexed Sequential file are always allowed. For the 9.0 version of the software, all logical update operations, WRITE and REWRITE, will be started only after all accesses to the file have been suspended. These update operations will inhibit further accesses to the file until they complete. To users, it will appear that concurrent updates to the file are allowed, though this will not actually be the case.

This restriction simplifies the code necessary to insure that the appropriate buffers remain in memory. Since only one update operation can be in process at any given time, the update operation will begin with a BD.USER.COUNT of zero. Once the update operation uses a buffer, that buffer's user count will be set to one, thus preventing the Memory Management algorithm from overlaying it.

Upon completion of the update operation all user counts will be set to zero. For READ operations, the user count field is not used because each buffer need be used only once during the process of the communicate. The buffer is automatically protected from being overlaid while the I/O operation is in process.

The code necessary to insure the integrity of the file is also simplified. The Record Contention problems, the complex problems involving changes to the file while another user is accessing it, are avoided. For the simple case of one user at a time updating the file, The simplified code provides better performance.

Disk I/O Error Procedures

The disk I/O error procedures in the MCP perform a certain number of retry operations each time a disk I/O operation completes with the Exception Bit, Bit 1 starting from zero, set to one. Different procedures may be invoked, depending upon the type of I/O operation that has completed and the type of control and drive that encountered the error. MCP I/O operations are handled by a different procedure which is not as extensive as the one described below. The following description applies to I/O errors on user I/O operations only.

The I/O error procedure first checks the Memory Parity Error bit, Bit 5 in the Result Descriptor, received from the control. If the bit is on, it performs a maximum of three retry operations,

logs the result and exits the procedure, without investigating any other bits in the result descriptor.

The procedure next checks the Transmission Parity Error bit, Bit 15 in the result descriptor. If this bit is on and if the unit being used is not a B9482 disk cartridge, the procedure performs a maximum of three retry operations, logs the result and exits the procedure without checking any further. If the unit is a B9482, no retry operations are performed for this case but and the investigation continues.

The procedure next checks the Not Ready bit, Bit 2 in the result descriptor. If this bit is on, the procedure performs a maximum of three retry operations, logs the result and exits the procedure without checking any further.

The procedure next checks the Write Lockout bit, Bit 6 in the result descriptor. If this bit is on, The procedure looks at the I/O descriptor itself. If the first three bits of the operation code are 010, 011 or 101, which would denote Write, Initialize and Relocate, the procedure performs a maximum of three retry operations, logs the result and exits the procedure without checking further. If the first three bits denote something other than the three operations listed, Bit 6 is ignored and the investigation continues.

The procedure next performs a Logical OR operation on:

1. The Sector Address Error bit, Bit 10,
2. The Seek Timeout bit, Bit 11,
3. (The Address Parity bit, Bit 9, AND not B9482), and
4. The Data Error bit, Bit 3.

If the result of the Logical OR operation is true, the procedure becomes complex and varies with the type of disk connected. Before describing the procedure for each type of disk, some basic procedures should be described.

The Offset Procedure

The Offset Procedure is a subroutine of the disk I/O Error procedure. Basically, it performs six retry operations. If any one of the six effect recovery of the error, the procedure is exited immediately regardless of how many operations have been performed. The term "offset" as used here denotes positioning the disk heads slightly off of the center of the cylinder specified in the disk address. In all disk pack drives which may be connected to the B1000 system, offset may be specified in the

inward (positive) or outward (negative) direction.

The first two operations requested by the Offset Procedure are performed with the original I/O descriptor unmodified. The next two operations are performed with negative offset and the last two are performed with positive offset. If recovery is not effected by any of the six, all bits which may have been set in the original I/O descriptor to cause the offset operations are reset and the procedure is exited.

The Strobe Procedure

The term "strobe" as used here denotes beginning the actual read operation sligtly before or after the point in the rotation of the disk where it would normally begin. The Strobe Procedure calls the Offset Procedure a maximun of three times. This may cause a maximum of eighteen retry operations to be performed. If any one of the eighteen effect recovery, the procedure is exited regardless of how many operations have been performed.

The first call on the Offset Procedure is accomplished with the original I/O descriptor in its unmodified form. This will cause six retry operations to occur, exactly as described for the Offset Procedure, provided recovery is not effected by any of the six. The next call is accomplished with a bit set in the descriptor which will cause early strobe to occur. Hence, another six retry operations may be performed, two with early strobe and no offset, two with early strobe and positive offset and two with early strobe and negative offset.

Twelve retry operations have been performed to this point. If the error has not yet been corrected, the Offset Procedure is again called with bits set in the I/O descriptor to cause late strobe to occur. This may result in another six retry operations being performed, as described for the Offset Procedure, all with bits set in the I/O descriptor to cause late strobing to occur.

If none of these eighteen operations effect recovery, all bits which may have been set in the I/O descriptor are reset and the procedure is exited. In the Strobe Procedure and in the Offset Procedure, if any retry operation does effect recovery, the I/O descriptor resposible is entered in the log prior to exiting the Procedure.

The Error Correction Procedure

All varieties of disk pack that may be connected to the B1000 system and some varieties of disk cartridge include error correcting capabilites in the form of a Fire Code remainder stored immediately after the 1440-bit data segment. The remainder is fifty-six bits in length on the 207 disk pack and thirty-two bits in length on all others. It is computed and stored by the disk hardware when the data segment is written. If an error should occur when the data segment is read, the data as it should have been written may be reconstructed, provided all of the bits in the data that are incorrect reside in the same "burst" of bits and provided the length of this burst does not exceed a specified limiting number of bits.

The Error Correction Procedure obtains a 2,080-bit buffer from available memory. If such memory is not available, the routine exits without attempting to correct the error. In all cases, when error correction is performed, all of the segments described by the original descriptor are read and corrected one sector at a time. For all disk devices which store the 32-bit remainder but which do not have the ability to correct burst errors in input data, the procedure must operate in this manner. Devices which are capable of performing error correction, such as the 207 disk drive, are capable of doing so on multiple-sector read operations, but this feature is not utilized by the software. Rather, all of the sectors are read one sector per operation and the exact addresses of all failed sectors are logged. This information would be lost on a multiple-sector read operation.

Error correction is performed by the software for all varieties of disk pack except the 207. The 207 hardware includes error correcting capabilities. Error correction is also performed by the software for the B9482 Disk Cartridge. The software is capable of correcting a six-bit error burst. The 207 hardware is capable of correcting an eleven-bit burst.

Data and Address Error Recovery - 215 And 225 Drives

Two different varieties of 215 and 225 disk pack drives have been delivered during the life of the B1000 hardware. These varieties are known as Design Level One (DL-1) and Design Level Two (DL-2). For both varieties, the Strobe Procedure is invoked but there are some operational differences in the hardware itself. On DL-1 drives, the bits which cause plus and minus offset and early and late strobing are ignored by the hardware, since it does not include these capabilities. Consequently, on DL-1 drives, a total of up to eighteen retry operations will be performed by the Strobe Procedure, but they will actually be nothing more than

eighteen repetitions of the original I/O descriptor. DL-2 drives include a full complement of offset and strobe capabilities. The software cannot distinguish between the two types of drive.

If none of the eighteen retry operations caused by the Strobe Procedure effect recovery, the I/O descriptor is restored to its original state, and the Error Correction Procedure is invoked. Each segment described by the I/O descriptor is read individually and error correction is performed, if possible, by the software. In all cases, the results of the recovery attempt are entered in the Engineering Log.

Data and Address Error Recovery - 205 And 206 Drives

For 205 and 206 disk drives, the Strobe Procedure is performed exactly as it is described. Eighteen retry operations are performed, two operations with each possible combination of the strobe and offset variants. If any of these operations effect recovery, the I/O descriptor is restored to its original condition and the procedure is exited. If not, the Error Correction Procedure is invoked with the I/O descriptor in its original condition. Error correction is performed by the software for 205 and 206 drives.

In any case, the results of the recovery attempt will be entered in the Engineering Log prior to exiting the procedure. The I/O descriptor is always restored to its original condition prior to exiting the procedure.

Data and Address Error Recovery - 207 Drives

207 disk drives include neither offset capabilities nor strobe capabilities. The hardware does include a capabilty to vary the threshold of a read operation but its use is not recommended for recovery purposes by the manufacturing plant. Consequently, the Strobe Procedure is not invoked for 207 drives. Two retry operations only are performed, both using the original version o the I/O descriptor. If either operation effects recovery, the results are logged and the procedure is exited. If not, the Error Correction Procedure is invoked.

207 drives include error correcting capabilities in the hardware. Additionally, the hardware is capable of correcting all errors that are correctable in all sectors described in one multiple sector operation. This multiple sector capability is not utilized by the software, however, and each sector is read and corrected individually. This is done for diagnostic purposes

only, to isolate the address of the failed sector(s) and insure their entry in the Engineering Log. The results of the recovery attempt will be logged and the procedure will be exited with the I/O descriptor restored to its original condition.

Data and Address Error Recovery - Disk Cartridges

For all versions of disk cartridge except the B9482, the 4400 BPI, 203 or 406 track, 64 sector per track variety of cartridge, the error recovery procedures are very simple. The procedure merely repeats the original operation a maximum of three times. The results of this attempt are logged and the procedure is exited with no further checking. There are no other options available in the hardware which might help in the recovery attempt.

For the B9482 cartridge, the recovery attempt is slightly more extensive. This drive has error correcting capabilities similar to those of the 206 drive. Error correction on a Read operation is performed by the software in the Error Correction Procedure exactly as it is described. On a Write operation, the recovery attempt is actually more complex than for a disk pack.

When a Write error occurs on the B9482 cartridge, the I/O Error procedure will attempt to correct the error, if the three retry operations mentioned above fail, by writing the data one sector per operation. In the case of an Address Parity error, the procedure will also attempt to write that sector plus the preceeding sector in an effort to correct the address parity. The results of the attempt will be logged and the procedure will be exited when recovery is effected or when all retry attempts have been completed.

This concludes the discussion of Data and Address Error Recovery for the various drives that may be connect. The remainder of this section describes the remaining tests in the I/O Error procedure.

Remainder of the Disk I/O Error Procedure

If the results of the Logical OR operation mentioned previously were false, the I/O Error procedure examines Bits 22 and 23 of the result descriptor. If both bits are set to one, they indicate that an Extended Result Descriptor was returned with the operation, though the ERD may not be stored in memory. The procedure stores the Extended Result Descriptor, if it is available, in the Engineering Log and performs a maximum of three

retry operations using the original I/O descriptor. The results of the attempt are logged and the procedure is exited with no further checking.

Finally, if all of the tests mentioned to this point were false, the procedure performs a maximum of three retry operations and logs the results. Since an exception did occur, indicated by the setting of Bit 1, the data is assumed to be corrupt and an attempt is made to correct it.

Tape I/O Error Procedures

There is one I/O Error procedure that is invoked for all tape I/O operations that complete with the Exception bit in the result descriptor set. The procedure is invoked regardless of whether the operation was a user I/O or an MCP I/O. It is also invoked on the completion of Test operations, where the setting of the Excepton bit is a normal occurrence. It is also invoked for Emulator Tape operations, though in this case, it may do nothing more than pass the result descriptor on to the user for resolution.

Essentially, the procedure will retry the operation a fixed number of times and return control to the procedure which called it. If recovery was effected, this will be so indicated in the previously failed result descriptor upon return. If the procedure was not able to effect recovery, the result descriptor will contain an indication of the failure upon return. In most instances, the procedure will retry the operation ten times, but this number will vary with the type of failure and the operation attempted.

The Tape I/O Error procedures will be described fully in a subsequent version of this specification.

S-MEMORY MANAGEMENT AND MEMORY REQUIREMENTS

This section of the specification has two principle parts. S-memory management is described at the functional level. S-memory requirements for a given system configuration are then presented. Using the second part of this section, it should be possible to estimate the amount of S-memory that will be required on a system to support a given program.

S-memory management techniques were changed drastically in the 7.0 version of the software and were changed again in the 9.1 version. The discussion contained in the first part of this section may not be applicable, in all cases, to versions of the software released prior to the 9.1 version.

GENERAL MEMORY MANAGEMENT CONCEPTS

The B1000 software utilizes a "segmentation" form of memory management. In such a system, memory is requested and allocated only when it is required and only in the amount that will exactly satisfy the request. In other words, memory is divided into a variable number of segments, each of which is of any size, with some obvious restrictions. A basic element in this form of memory management is the "memory link".

The format of the memory link was presented in a prior section. Basically, it contains a size field which may contain any value from zero to 16,777,215 bits. It contains the addresses of the memory links that precede and succeed it and the address of an associated segment dictionary entry. It contains a number of other fields, which will be discussed in turn. It is created and maintained by the MCP and the executing interpreters store selected information in it. In all cases, it immediately precedes the segment of memory that it describes.

LINKED MEMORY

Contiguous blocks of memory are reserved for system use at the extreme ends of the memory on any system. This is described in more detail in the second part of this section. Between the two contiguous blocks lies the area known as "linked memory". At the end of the reserved area at the low end of memory, there is a dummy memory link known as the Lower Terminating Memory Link (LTML). At the beginning of the reserved area at the upper end of memory is the Upper Terminating Memory Link (UTML).

The terminating memory links are created during the Clear/Start procedure. Each has a size field of zero, a type field which specifies the area as TERMINATING.LINK, but the "save" bit will be set to one in both links. This allows the memory management procedures to recognize the terminating memory links. The backward pointer in the LTML will contain #FFFFFF#; but the forward pointer will contain the address of the next memory link, in address order. Similarly, the backward pointer in the UTML will contain the address of the previous memory link in address order; the forward pointer will contain #FFFFFF#.

Hence, all memory links form a chain in memory. The memory link which immediately precedes each allocated memory area will contain the address of the succeeding and preceding memory links in the forward and backward pointer field respectively. The chain will be terminated in the forward direction by the upper terminating memory link and in the backward direction by the lower terminating memory link.

The area known as linked memory is an example of a "memory subspace", as this term is used herein. There may be other memory subspaces within linked memory. The Run Structure (Base/Limit area) of certain programs may also be divided and allocated upon request by the software. The same procedures in the software are used to manage these smaller memory subspaces as are used to manage linked memory.

TYPES OF MEMORY REQUESTS

Memory requests may originate in a number of diverse manners. This is evidenced by the large number of different values the type field of a memory link may contain. The most common occurrence of a memory request is for a code segment to be brought into memory. Other requests originate when a file is opened, when the MCP needs additional temporary storage for the performance of one of its tasks, when additional space is required to hold a queue message, and so forth.

There is probably no need to discuss each different type of memory request. Many of the numbers assigned to each different type of memory request are for the benefit of the Dump/Analyzer program only and have only pathological use. The different types of requests have common characteristics and may hence be grouped into "classes". The common characteristics will be described and explained.

Parameters that are passed with a memory request are the size of the required memory area in bits, the address of the dictionary

entry which will be associated with the memory area, if any, the address of the Run Structure Nucleus of the program which caused the request, if any, the type field to be stored in the memory link, the priority of the request, a boolean variable which specifies that the memory should be allocated at the highest possible physical address and a boolean which specifies that the memory must be allocated above the "fence".

THE FENCE

The MCP has one set of stacks, only, to store the variables that it must manipulate in the performance of any function. This set of stacks cannot be stored anywhere else; they must be maintained in memory until the function has been performed. Consequently, once the MCP begins performing any function, it can perform no other function until the original task is complete.

Almost all MCP functions require more than one MCP code segment to complete. A file Open may require more than thirty code segments to be brought into memory. The number of code segments required could obviously be reduced by making each code segment larger but this would also reduce the possibility of finding sufficient memory on small systems. It would also possibly cause more user code segments to be removed from memory to make room for the larger MCP segment.

Should the MCP begin performing a certain request and not be able to find sufficient memory to contain a necessary code segment, the system would have to halt. A Clear/Start would then be required, with its resulting loss of all programs that were running at the time. In order to insure that there will always be sufficient memory to bring in the largest MCP code segment, a fence is established in memory, below which only code segments are allowed to reside. The location of the fence may be calculated by adding the size of the largest MCP code segment and its associated segment dictionary to the address of the lower terminating memory link.

Certain exceptions to the statements in the paragraph above exist. Code segments may not be overlayable at all times. To bring a code segment into memory, the memory area is allocated and an I/O operation initiated. The memory area may not be deallocated until the I/O operation is complete. Should the MCP encounter such a situation and not be able to find a required memory area anywhere else in memory, it will wait for the completion of the operation.

Certain code segments associated with MICR applications programs are also not overlayable. This is also true of segments of the interpreter used by such programs. Consequently, the fact that a memory request is for a code segment is not sufficient to determine whether the memory should be allocated below the fence and the boolean variable is required.

MINIMIZATION OF "CHECKERBOARDING"

Checkerboarding, also known as External Fragmentation, is the condition which exists when memory contains a large number of permanently-allocated areas, or "save" areas, most of which are separated by small overlayable areas. In such a situation, the total memory available may well be large enough to satisfy a given request but no single contiguous overlayable area is sufficiently large. This situation can have a serious impact upon performance.

To minimize the possibility of the occurrence of checkerboarding, the MCP attempts to allocate all memory denoted as non-overlayable or "save" at the highest possible physical address. Examples of items which are so allocated are program run structures, files and I/O buffer areas.

VICTIM SELECTION

When a request for memory allocation is received, the management algorithm must select a "victim", a portion of memory which is already allocated which may be deallocated and assigned to satisfy the new request. The area to be allocated may also be marked Available, of course, though this is seldom the case. "Victim Selection" is the process of determining which allocated memory segment or segments will be deallocated. This is the most intricate task of the management algorithm, the task which requires the most attention to strategy and the task which is most influential upon the performance of the system. Two victim selection algorithms are provided in the software. Users may choose either the priority Victim Selector or the Second Chance Victim Selector via a system option. The change is only effeccted during a Clear/Strat operation.

ROUND-ROBIN VICTIM SELECTION

Prior to the 7.0 release, victim selection was essentially a round-robin among requests. The MCP kept a pointer which served as the starting point of each search and was updated after each allocation to point to the end of the newly allocated area. This pointer is typically referred to as the "left-off pointer". The round-robin algorithm had the advantages of being computationally simple and it served to minimize external fragmentation, but there are some serious disadvantages associated with this victim selection algorithm. Specifically,

1. It has no knowledge of which segments are actually in use, elements of the "working set", and

2. The memory resources of each job have equal importance. Unlike processor scheduling, the memory is not allocated on a priority basis.

These flaws lead to some bad performance degradations in certain situations. One such problem is the "cascading" phenomenon.

Using Denning's definition, a program's working set W(T, t) is the set of all segments accessed by the program in the interval [T-t, T]. Denote the size of this set (in MCPII context, size is in bits), as W(T, t). This definition affords us useful information with which to manage real store whenever the change ("drift") from the set W0(T0, t) to the set W1(T1, t) is small for the interval [T0, T1]. The assumption behind working set management is that for many programs, the drift is indeed small during most of their execution lifetimes.

Postulate a situation where the code and dictionary segments of a single job completely fill overlayable memory. The round-robin algorithm, having no information concerning W(T, t) made a choice of victims among the resident segments which was essentially random with respect to this information. Call the ratio W(T, t)/(size of overlayable memory) the saturation ratio S. Then the probability is approximately S that the incoming segment will overlay one or more elements of W(T, t). The overlayed segment, of course, will immediately be needed again and has a probability of about S of overlaying another element of W(T, t). This sets up an undesirable oscillation which should eventually damp back to stability, assuming no further external perturbances. The probable number of extra overlays required to reach stability increases with S, and becomes quite large when S exceeds, say, 0.9. We call this oscillation "cascading" of overlays. For large values of S, almost all time is consumed in waiting for I/O on the backing store, so very little work gets done. This is the situation commonly known as "thrashing".

Now, suppose the memory manager has some knowledge of the elements of W(T, t). If the saturation ratio is not too close to one, it will usually be possible to select a window containing no element of W(T, t). The chance of cascading segments is thereby decreased in configurations running with S in the range of 0.5 to 0.75. The difficulty is that elements of W(T, t) now clutter memory and increase external fragmentation. As S approaches (or exceeds) one, this becomes an important loss and makes selection difficult for the memory manager. At this point, the advantages of the round-robin strategy begin to outweigh the advantages of utilizing working set information.

WORKING SET DETERMINATION

In order to determine whether or not a code segment in memory is currently being used, usage bits were added to the memory link in the 7.0 version of the software. These appear in the programmatic description of the memory link as ML.PREVIOUS.SCAN.TOUCH and ML.CURRENT.SCAN.TOUCH. Whenever an interpreter accesses a code segment dictionary entry and finds the associated code segment present in memory, it sets the current scan touch bit to a value of one. Interpreters make such an access whenever they are reinstated and whenever a code segment transition occurs. It is not necessary for interpreters to set the bit in memory links which are associated with segment dictionaries. These are usually marked as save space if any of their code segments are present in memory. Also, data segments are always overlaid in a round-robin fashion, regardless of the victim selector that is currently being used on a system.

SECOND CHANCE VICTIM SELECTION

The Second Chance victim selection algortihm, first introduced in the 9.1 version of the MCP, addresses the first failing of the round-robin algorithm, the lack of knowledge of the working set of the code being used. Also, the Second Chance algorithm completely supplants the old round-robin strategy. The latter is no longer available for use. The change is completely transparent to users and the only noticeable effect should be an improvement in performance in installations where the round-robin algorithm was used prior to release of the 9.1 software.

The Second Chance algorithm utilizes the left-off pointer described for the round-robin algorithm. It begins searching for a memory space large enough to satisfy the request at the left-off pointer but it will not select any space whose touch bit, ML.CURRENT.SCAN.TOUCH, is set. Upon encountering a memory

segment whose touch bit is set, it resets the bit and continues to the next memory link. It will allocate the first segment it encounters that is sufficiently large and whose touch bit is reset.

This algorithm thus has the major advantage of the round-robin algorithm; it is computationally simple and the procesing required is minimized. Unlike the Prioritized victim selection algorithm described below, it requires no knowledge or action on the part of the user.

PRIORITY VICTIM SELECTION

The second failing in the round-robin strategy is its inability to insure rapid turnaround to jobs which are designated as high priority. In MCPII, prior to the 7.0 release, only the processor was allocated on the basis of priority. A high priority application was contending for the memory resource on exactly the same footing as a low priority "background" job. This led to severe performance degradation for users which required many overlayable memory resources but frequently relinquished processor control to make operating system requests. In particular, datacomm applications running in multiple job shops were suffering badly. Background jobs tended to usurp critical resources forcing the datacomm application to loose control still more frequently, allowing background jobs to run, grab more memory resources, and so forth.

The Prioritized memory management algorithm, first introduced in the 7.0 version of the MCP, addresses both of these problems. The priority victim selector makes its choices on the basis of a priority field in each memory link. This field is maintained by runtime use of working set information. The priority field will be maintained at its original value as long as the code segment is not used. This field is known as the Residence Priority field and is shown on the programmatic memory link description as ML.RESIDENCE.PRIORITY.

Associated with each program running on the system is a Memory Priority field. The memory priority value determines the ability of the program's code segments to overlay the code segments of other programs running on the system. Memory Priority is stored in each memory link associated with each of the program's code segments. It is shown programmatically as ML.INCOMING.PRIORITY. Memory priority is also stored initially in the Residence Priority field. Whenever a request for a new code segment to be brought into memory is received, the memory priority of the associated program is compared to the residence priority of every memory link currently present in the system memory. The current

implementation of the victim selector always chooses a victim having the lowest residence priority.

An exception must be made for MCP code segments. As presented in a prior paragraph, the MCP cannot be denied a requested code overlay without halting the system. Consequently, MCP code segments have an imperative incoming priority, but their residence priority value will decay at a rate equal to or greater than the programs running on the system.

At a user-specified interval, a routine in GISMO known as the sweeper is executed. This routine moves the setting of the current touch bit to the previous touch bit, destroying the prior setting of the previous bit and setting the value of the current touch bit to zero. This routine is discardable and is eliminated by the initializer if the system is running with the Second Chance victim selector.

The default time period between executions of the sweeper is 800 milliseconds. Users may vary this time period via a keyboard instruction within certain ranges. Since the sweeper routine may be executed between any two S-operations, all code in the software which manipulates memory links must always insure that the chain formed by the address pointer fields is intact.

After the sweeper has moved the current touch bit to the previous touch bit, it then examines the previous touch bit. If the value is zero, it increments the current decay interval field, ML.CURRENT.DK.INT, by the value of the sweep interval. If the value of the current decay interval is equal to or greater than the specified decay inteval, ML.DK.INTERVAL, the residence priority field is decremented.

The default value of the specified decay interval is zero. Users may specify different decay interval values via a keyboard instruction. Users may also specify that certain code segments within a program are important and that their residence priority should not decay until the specified decay interval has elapsed. This is accomplished via a supplied normal-state program which manipulates code files resident on disk. The residence priority of code segments which are not marked as important will decay after the default decay interval, zero seconds, has elapsed. Notice, however, that this cannot occur for at least one sweep interval.

When executing with the priority victim selector, the MCP still maintains a left-off pointer. When the system is thrashing, when the residence priority fields of all memory links have equal

values. the victim selected will continue to be the next memory area below the left-off pointer.

PROGRAMMATIC DETECTION OF MEMORY THRASHING

One of the serious problems confronting virtual storage systems is memory thrashing. On the B1000 system, memory thrashing occurs when the working set of procedures for a program or set of programs will not fit within the portion of main memory available for overlays. When this state occurs, the system's performance begins to degrade. The amount of degradation depends on the overlay space available, the size and number of segments competing for memory, and the frequency of segment transitions.

As the amount of main memory is reduced for a constant programming task, the amount of degradation due to memory overlays normally appears very gradual at first. As the available memory is further reduced, a point will be reached where the degradation due to overlays increases rapidly. This is the point where the main working set of procedures no longer fit in main memory and are competing for space. This point is defined as the thrashing point and is shown in Figure 4.1.

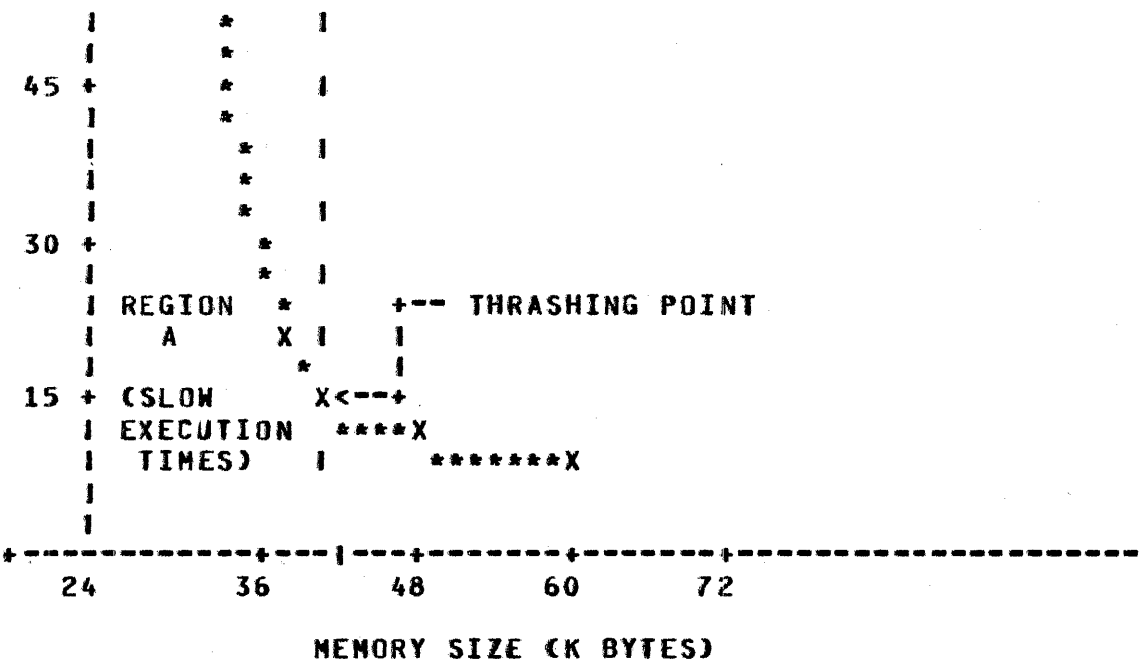


FIGURE 4.1: MEMORY VS EXECUTION TIME

EXAMPLE PROGRAM ARM020

As seen in Figure 4.1, execution of the programming task with less memory than indicated by the thrashing point yields inefficient execution times in region A.

Beginning with the 7.0 version of the software, the MCP includes a programmatic facility for detecting a thrashing condition in the system. The facility is included in GISMO as a discardable segment; it is retained or discarded during the Clear/Start operation based upon the setting of a system option. It may be used with either victim selection algorithm. It must be used if the priority victim selection algorithm is used.

The facility is actuated by a clock maintained in the software. It utilizes a count of the number of overlay operations performed by the software. The count is also maintained in the software, of course. The sweeper routine discussed previously is actuated by the same clock that actuates the thrashing detection routine.

At a user-selected interval, the thrashing detector compares the number of overlays which occurred during the interval to a user-specified target number of overlays. If the overlay target is exceeded, the thrashing detector suspends temporarily the execution of the sweeper routine and begins a count of the number of consecutive intervals during which the number of overlays exceeds the target number. The allowable number of intervals during which thrashing, as defined by the user, is detected is three.

If the thrashing condition persists for three intervals, the software informs the operator via a SPO message. The message will be repeated at N.SECOND intervals until the condition abates or until the operator requests, via another SPO message, that it not be displayed continually. The software also disables the schedule when thrashing is detected so that no new jobs are initiated. The schedule will be automatically enabled again when a program currently being executed terminates.

MEMORY INITIALIZATION

Memory is initially allocated by the software during the Clear/Start operation. This single operation is composed of several components. For discussion purposes, it may be thought of as two separate operations. The first of the two is the execution of a stand-alone routine, commonly known as the Initializer and stored in the disk directory as SYSTEM/INIT. The initializer is brought into memory by the Clear/Start code contained on the cassette. The second operation is the execution of some code in the MCP, contained in Page Zero, Segment One of

the MCP's code file.

At the completion of the initializer, memory will be formatted as shown in Figure 4.2. Permanently allocated areas will be located at each end of memory. Linked memory will consist of four links only. The processor is then passed to the MCP's code segment for completion of the Clear/Start operation. Upon completion of the MCP code, linked memory will be formatted as shown in Figure 4.3.

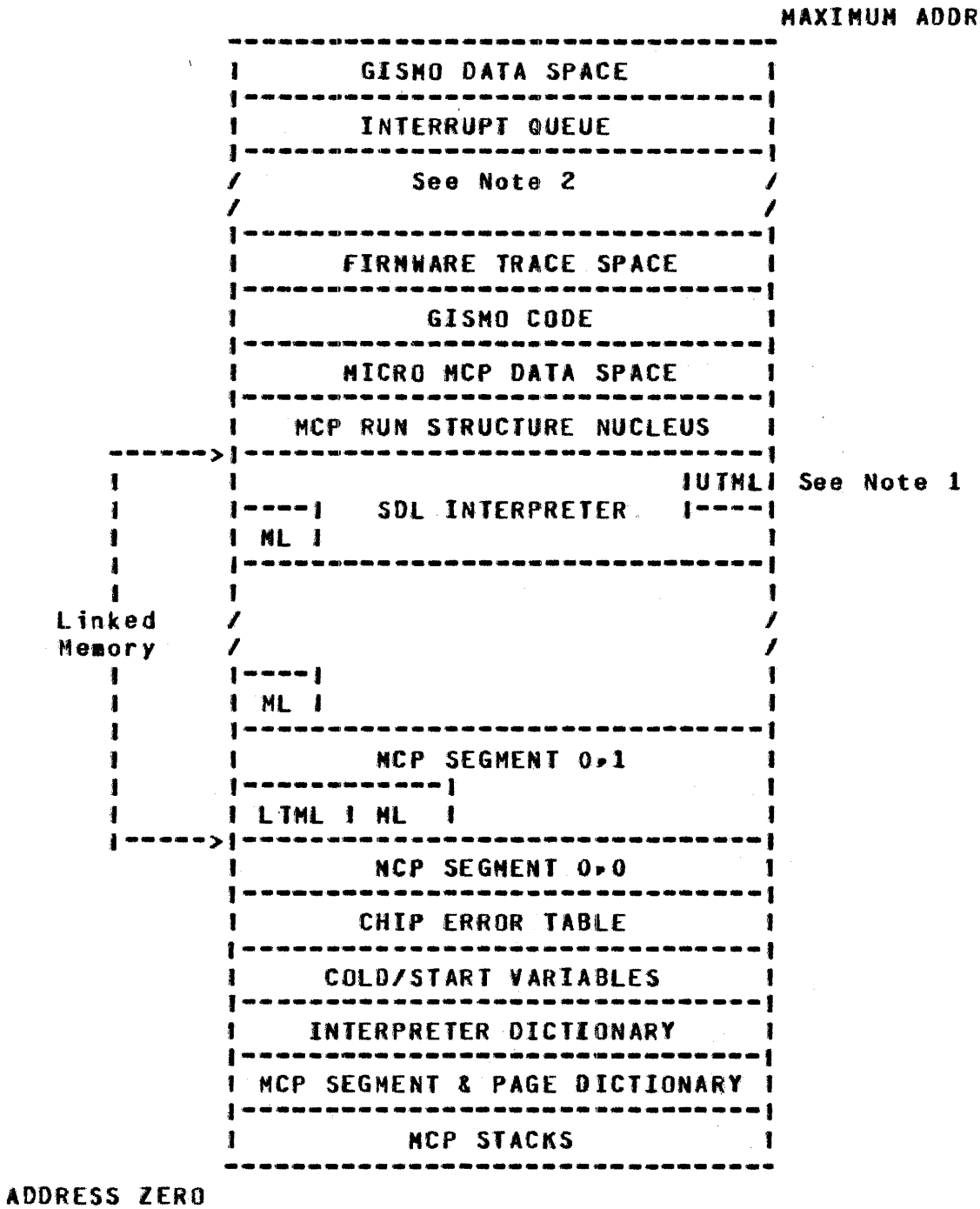


Figure 4.2 Memory Format After Initialization

Figure 4.2 Notes

1. "UTML" and "LTML" are acronyms for upper and lower terminating memory links. These two links have a size field of zero and a type field which denotes a terminating memory link. The upper link has a forward pointer of @FFFFFF8; the lower link has a backward pointer of @FFFFFF8. The links are used to mark the boundaries of linked memory for the memory allocation routines. Memory allocated by these routines will always lie between these two links.

2. It is possible, during the initialization procedure, for the operator to specify a maximum S-memory address that is less than the actual maximum address of memory on the system. When this is done, a proportionate amount of memory is reserved at the location shown. This memory is, in effect, deleted from the system. Memory may also be deleted via certain keyboard instructions available to the operator. In the latter case, the deleted memory may lie at almost any address in the system.

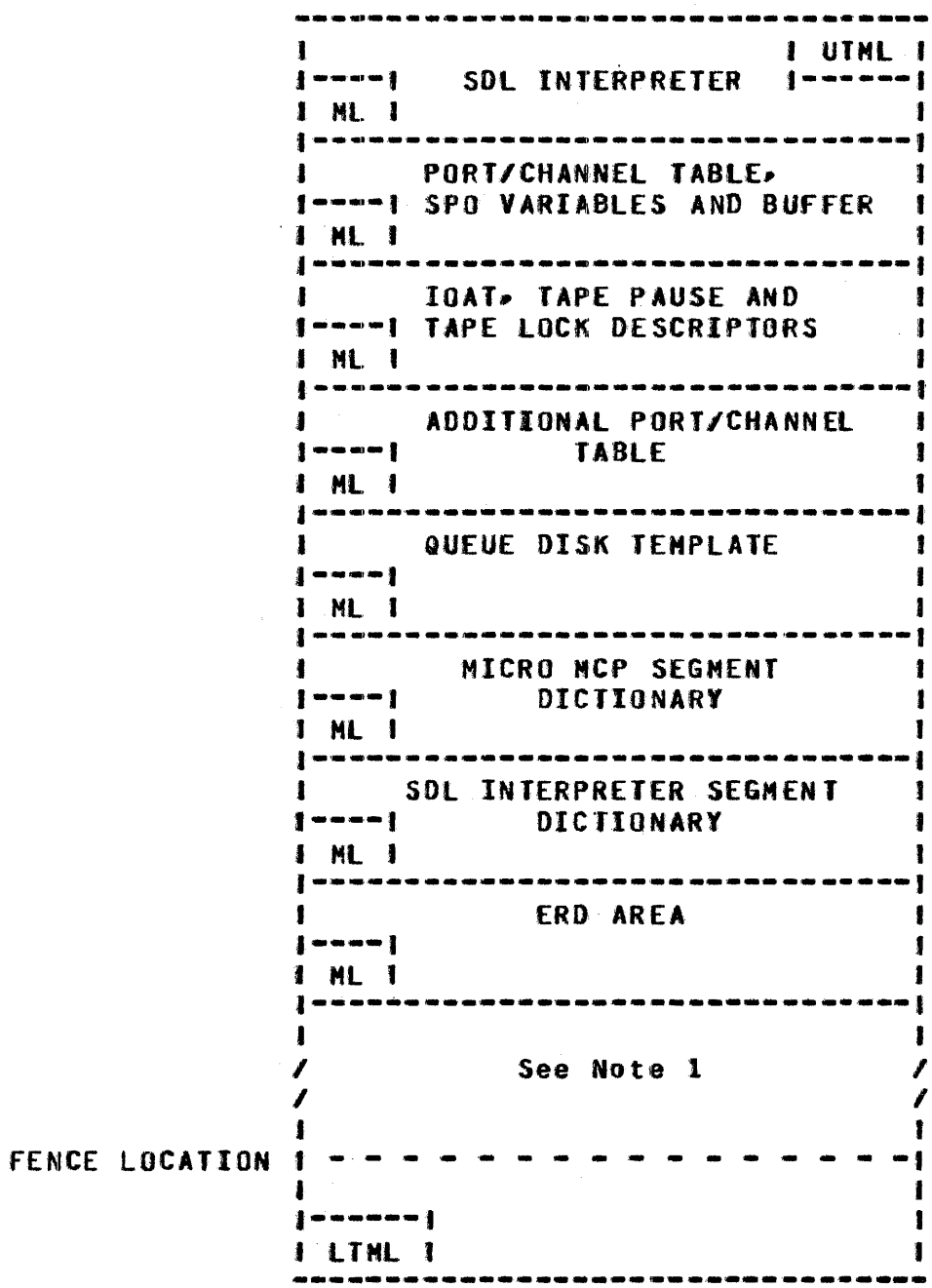


Figure 4.3 Linked Memory Format After Clear/Start

Figure 4.3 Notes

1. Though nothing is shown as present in figure 4.3 between the SDL Interpreter Segment Dictionary and the Lower Terminating Memory Link, this area will typically be filled with MCP code segments at the completion of the Clear/Start operation.

2. The purpose of the fence shown in figure 4.3 was discussed previously. The location of the fence is retained in the MCP stacks. It is not necessary to reserve any memory at all at the fence location.

MEMORY REQUIREMENTS

The memory that will be required to execute a given program or set of programs is composed of four components. There are the static requirements of the operating system, the dynamic requirements of the operating system, the static requirements of the program and the dynamic requirements of the program.

Static requirements are composed of the data spaces necessary to operate the system and the program. Once the static requirements are established, they typically do not change. For example, once a program has all of its files open, the memory required for the File Information Blocks and the buffers remain fixed until the files are closed. In the case of the MCP, once the system is Clear/Started, the static requirements remain fixed until the system is Clear/Started again.

Dynamic requirements are exclusively code segments. Assuming that a working set of the code segments of a program is established, the dynamic requirements for that program will then be the total amount of memory that is required to contain the code segments that are a part of the working set. The operating system's working set depends, of course, upon the communicate operators that are issued by the program in its own working set.

OPERATING SYSTEM STATIC REQUIREMENTS

Those items shown in figures 4.2 and 4.3 comprise the static memory requirements of the operating system. Each item will now be discussed and a means of determining the amount of memory required by the item will be presented. The numerical values presented herein apply to the 9.0 version of the MCP only.

MCP Stacks

The stacks used by the MCP will always reside at location zero in S-memory. For each released version of the MCP, the stacks will be of a fixed size, regardless of the machine configuration. The stacks require roughly 34,416 bits or 4302 bytes.

MCP Page Dictionary and Segment Zero Dictionary

These two items may never be overlaid and are maintained in memory immediately above the MCP stacks. They will also be of a fixed size for each released version of the MCP. The 10.0 version of the MCP code is divided into thirty-four segment pages and Page Zero contains thirteen segments. Each entry consists of a system descriptor, which requires 80 bits or 10 bytes. For the 10.0 version of the MCP, this item requires 3760 bits or 470 bytes of memory.

Interpreter Dictionary

An Interpreter Dictionary entry requires 224 bits or 28 bytes. The number of interpreters that may be used on a system at any given time, and hence the number of entries allowed in the Interpreter Dictionary, may be specified by the user to be any value between 3 and 31. If the user does not specify this number, the Cold/Start routine will set this value to six. The memory required for the Interpreter Dictionary may be calculated by multiplying the number of interpreters allowed by the size of one entry.

Cold/Start Variables

The variables contained in this area are originally set by the Cold/Start routine. Many of them may be changed by the operator. The memory allocated for their storage may not be changed. It will also be a constant value for each version of the operating system. For the 10.0 version, the memory required is 2256 bits or 282 bytes.

Chip Error Table

This area is allocated on B1800 and B1900 series machines only. On all other machines, no memory is required for this item. On the B1800 and B1900, the area is used to store the addresses of memory locations which are experiencing correctable memory parity errors. The size of the area in bits may be calculated by 40 plus (32 times the number of entries allowed in the table). The operator may specify the number of entries the table should contain. The default

value for the number of entries will be one entry per 16K bytes of S-memory on the system.

MCP Code in Page Zero, Segment Zero

This code segment is normally referred to as "Segment Zero" and the size of the segment is a constant for each released version of the MCP. This is the only MCP code segment which does not require a memory link, since it is outside of linked memory. The code segment requires roughly 53,490 bits or 6686 bytes of memory.

Upper and Lower Terminating Memory Links

In the 10.0 verson of the software, a memory link requires 187 bits of memory. These two then require 374 bits or 47 bytes.

SDL Interpreter

The sizes of the SDL Interpreters presented here are for reference only. Accurate size figures and figures for the various segments of the interpreters are provided in the appropriate product specification. Segment Zero of the S-Processor version of the SDL Interpreter requires 8166 bytes. The same segment of the M-Processor version requires 8024 bytes.

MCP Run Structure Nucleus

The MCP requires a Run Structure Nucleus field as does every other program which executes on the system. For the 9.0 version of the software, 2386 bits or 298 bytes are allocated for this field.

Micro MCP Data Space

Currently, 1249 bits or 156 bytes are allocated for this space. This requirement is a constant and is not dependent upon machine configuration nor system options selected, but a dual processor configuration will require two such spaces.

GISMO Code

GISMO is not segmented. Selected portions of the GISMO code are "discarded" by the Initialization routine if they are not required on a given system configuration with a given set of system options selected. The amount of memory that will be

required to contain GISMO must therefore always be calculated.

The Main Block of GISMO requires 5500 bytes of memory. No memory link is required. The amounts of memory shown in the following table should be added if the condition specified is true.

System equipped with Memory Base 5	104 bytes
Processor is a B1830	436 bytes
Processor is B1720 series	540 bytes
Processor is B1860 series	642 bytes
Processor is Dual B18XX	1070 bytes
Reference address check option set	138 bytes
Thrashing detection option set	142 bytes
Prioritized memory management option set	364 bytes
TOUT option set	100 bytes

In the list above, the cassette device on the processor console is not considered a peripheral. Neither the cassette peripheral segment nor the magnetic tape or cassette segment should be added due to the console cassette.

The control exchanges segment should be added when the system is equipped with two or more disk or tape controls and the controls address the same peripheral units. High-speed controls are all disk pack controls and any controls which address phase-encoded tape drives. Under no conditions is it necessary to add any GISMO code segment more than once. The Dual Processor segment and the B1860 segment must both be added if the system is a dual processor version.

Firmware Trace Space

This area is allocated only when running with trace versions of the SDL Interpreter. It should never be allocated in a customer's machine. It requires 1,440 bits.

Interrupt Queue

Since interrupts occur asynchronously on the B1000 system, they must be queued until they can be handled by the appropriate operating system routines. One entry in the interrupt queue requires thirty-six bits. Forty-two bits are required for pointers and counters. The number of entries which may be queued on a given system depends upon the amount of memory on the system. The number of entries that will be allocated may be determined from the following table.

S-Memory on System	Entries

Less than 64K bytes	16
At least 64K bytes but less than 96K bytes	20
At least 96K bytes but less than 128K bytes	25
128K bytes or more	30

The smallest amount of memory that will be allocated for the interrupt queue is then (42 + (16 X 36) or 618 bits. The largest amount is 1122 bits.

GISMO Data Space

The GISMO data space is a work area required by GISMO. It is a fixed size and amounts to 376 bits.

DCPU DATA SPACE

This is also a work area. It is required on all dual processor machines and requires 350 bits.

Looking now at figure 4.3, the MCP, prior to completing the Clear/Start operation, will allocate space for those additional items shown on the figure. The location of the "Fence" is not important to the discussion of the memory requirements of the MCP. The fence is merely a means of guaranteeing that the MCP will always find space for its own purposes when such space is needed. The system would be forced to halt if the MCP could not find the space required.

All of the items shown in figure 4.3 reside in linked memory. One memory link (187 bits) is required to describe each of the items in figure 4.3

Extended Result Descriptor Area

One extended result descriptor, I/O descriptor and buffer is required for each 5N head-per-track disk control and for each disk pack spindle on the system. Each descriptor and its associated buffer requires 256 bits. This requirement applies to all disk pack drives interfaced to the B1000 system but not to cartridge drives. The memory required, in bits, may be calculated by 256 X (5N controls + disk pack spindles) + memory link.

SDL Interpreter Segment Dictionary

The segment dictionary of the SDL Interpreter is considered non-overlayable, since it contains a descriptor for segment zero of the interpreter which must be non-overlayable to execute segment zero of the MCP. The size of this area, in bits, may be calculated by 64 plus (80 times the number of segments which comprise the interpreter) plus the space required for one memory link. The SDL Interpreter contains 6 segments, plus Segment Zero.

Micro MCP Segment Dictionary

This segment dictionary is also considered non-overlayable. Its size may be calculated in the same manner as the size of the SDL Interpreter segment dictionary, 64 plus (80 times the number of segments) plus space for one memory link. The Micro MCP contains 18 segments, plus Segment Zero. The segment dictionary therefore requires 190 bytes.

Queue Disk Template

The MCP reserves 500 segments of system disk for its own temporary use. The address of this reserved area of disk, known as Queue Disk, is stored in the memory area known as the Queue Disk Template. This memory area will also contain one bit to denote the availability of each of the 500 segments, a 24-bit field which will be used to store the memory address of the next Queue Disk Template if an additional 500 segments must be allocated and a 128-bit field known as the Communicate Splitter Mask. This latter field is used to determine which communicate operations may be handled by the Micro MCP. The size of the initial Queue Disk Template field is therefore, 500+36+24+128 or 688 bits. Additional Queue Disk Template fields, if required, will occupy 560-bit areas. One memory link is required on each Queue Disk Template allocated.

Additional Port/Channel Tables

The MCP and GISMO communicate in a number of ways. One such way is the Port/Channel table. One Port/Channel table is allocated with the SPO variables and buffer at the high end of linked memory. If the system is equipped with multi-line controls, an additional Port/Channel table will be required for each one. A Port/Channel table requires 768 bits of memory plus the space required for one memory link.

IOAT (I/O Assignment Table)

Several items are grouped together in the space reserved for the IOAT. The IOAT itself requires one entry of 512 bits for each peripheral unit connected to the system with the exception of the SPO. Each disk pack spindle is considered a peripheral unit. Head-per-track disk is not. Data communications devices are not considered peripheral units for the purpose of calculating IOAT size, but each single-line control connected to the system requires one IOAT entry. One "Pause" descriptor, requiring 96 bits of memory, is required for each tape control, cassette control and MTC-2/MTC-4 exchange on the system. One "Lock" descriptor, requiring 168 bits of memory, is required for each tape and cassette unit connected to the system. One I/O descriptor of 248 bits is required if any number of flexidisk units are connected to the system. One memory link is required to describe the area containing these items.

Port/Channel Table, SPO Variables and Buffer

Information which the MCP needs to perform its function which is primarily concerned with the system SPO but also includes information on other aspects of the system is maintained in the area known as SPO Variables. This information requires 1351 bits of memory. The Port/Channel table requires 768 bits and the SPO buffer requires 560 bits, for a total of 2679 bits. One memory link is required to describe the area.

OPERATING SYSTEM DYNAMIC REQUIREMENTS

The operating system's dynamic memory requirements are determined solely by the size of the code segment which performs the functions requested by the user in the working set of his program. In determining this requirement, it is necessary to know what the program in question is doing. While programs could be and are written which have file open and close operations as a part of their working set, this is not normally the case. The vast majority of programs request only those functions which are micro-coded and included in the Micro MCP in their working set code. This statement is not true for programs which use DMS.

This document will not present the memory requirements for programs which use DMS. This information will probably be added at some point in the future, but for the present, only the code segment sizes for operations believed to be common and exclusive of DMS operations will be presented.

The list below presents a brief description of the function and the memory requirement for each of the Micro MCP segments.

SEGMENT.ZERO - 2306 Bytes

Segment Zero of the Micro MCP is always required in memory when programs are executing.

SERIAL - 1960 Bytes

This code segment handles reads and writes on serial files that are opened input or output but not in any combination form, such as input-output. Also, some files assigned to data recorders may not require this segment.

SEQUENTIAL - 762 Bytes

This code segment handles reads and writes on sequential disk files that are opened input-output.

RANDOM - 944 Bytes

This code segment handles reads, writes and seeks on code segments whose access mode is random. This code segment is required for all random disk files, even if the access mode is delayed random.

COMP.WAIT - 1136 Bytes

This code segment is required to handle complex wait communicate operations. All data communications handlers generated by the NDL compiler require the complex wait code to be present.

DATA.RECOR - 344 Bytes

This code segment is required to handle reads and writes on files which are assigned to data recorders and which are opened input-output or input with stacker selection capabilities requested.

HI.PRI.AND - 1292 Bytes

This code segment is required to handle all communicate operations on files which are assigned to reader-sorters.

QUEUE.READ - 856 Bytes

This code segment handles read and write operations on queues. Please refer to the paragraph at the end of this list.

PQM.GQM - 2674 Bytes

(Put Queue Message.Get Queue Message). This code segment handles reads and writes on files assigned to queues and to remote files. Please refer to the paragraph at the end of this list.

REMOTE.WRI - 2300 Bytes

This code segment is required to handle writes on files assigned to remote files. Please refer to the paragraph at the end of this list.

REMOTE.REA - 2890 Bytes

This code segment handles reads on files assigned to remote files. It is also required to handle many NDL/MACRO communicates. Please refer to the paragraph at the end of this list.

DC.INITIAT - 410 Bytes

This code segment handles the DC.INITIATE.IO communicate operation. This communicate is issued by all data communications handlers generated by the NDL compiler.

MESSAGE.CO - 208 Bytes

This code segment is required to handle the message count communicate operator, also issued by all data communications handlers generated by the NDL compiler.

VARIABLE.L - 412 Bytes

This code segment handles read and write operations on tape and disk files which use variable-length records. It is required in addition to the SERIAL code segment.

EMULATOR.T - 508 Bytes

This code segment is required to handle communicate operations requested by any emulator interpreter on files assigned to tape.

DELAYED.RA - 592 Bytes

This code segment, in addition to the random code segment, is required to handle reads, writes and seeks on files whose access type is delayed random. Emulator disk files are in this category.

INDEXED.SE - 3020 Bytes

This segment is used for I/O operations on Indexed Sequential files, first introduced in the 9.0 version of the software and described in the section of the document on the I/O Subsystem.

RELATIVE - 3638 Bytes

This segment is used for I/O operations performed on Relative files, also described in the I/O Subsystem section.

IPC.CODE - 568 Bytes

This segment is used to perform Inter-Process communication, a part of the ANSI '74 COBOL implementation first included in the 9.0 software.

All code necessary to handle queues, remote files, the DC.INITIATE.IO communicate and the MESSAGE.COUNT communicate are included in the Micro-MCP. Microcoding these functions resulted in some substantial performance improvements for most data communications applications. There are several reasons for the improvement, the most obvious being the greater efficiency of the code. Another factor is that a minimal amount of state information must be saved when communicating with the Micro-MCP.

A third factor is the elimination of the "bottleneck" problem, as it has come to be called, for data communications applications. This problem arises from the fact that MCPII is a flat structure and is capable of performing one thing at a time only. In other words, once the MCP begins performing an open request for example, it can do nothing else until it completes the open. An

open, of course, requires many accesses to the disk subsystem and the MCP must wait on the completion of each one. Normal-state programs are free to execute while the MCP is waiting on each access, provided they do not request an MCP service which must be handled by MCPII.

Consequently, user programs may now use the queue subsystem and the other items mentioned above while MCPII is servicing another request for other users. In previous releases, these same user programs had to wait until the MCP completed servicing the request it was working on at the time. Unfortunately, however, all requests for functions in the queue subsystem are not handled by the Micro-MCP. Many of them, and possibly all of them, may still be handled by MCPII.

All memory management functions are still handled by SDL code in MCPII. Any queue request which involves memory management will therefore have to be handled by MCPII. This will most often occur in situations where the available memory on a system is limited. Queue buffers may be written to disk by MCPII, and hence removed from memory, whenever the MCP needs space for something else. This will cause MCPII to be invoked when a program attempts to read a queue entry from that buffer.

Further, if a producer of queue entries fills an entire buffer before the consumer can empty it, a new memory buffer will be required. MCPII will be invoked to accomplish the allocation. Unfortunately, in both of these instances, the entire working set of Micro-MCP queue handling segments will be brought into memory, only to determine that SDL MCP segments are ready required. This can result in substantial performance degradation, particularly on systems where available memory is limited.

The situation described can be avoided, of course, by insuring that the consumer of queue entries removes them from the queue at the same rate that the producer enters them. Since it is only rarely possible for the programmer to insure that synchronization exists, a system option has also been provided in the 6.1 release which will insure that all queue requests are handled exclusively by the SDL MCP. By setting the option, the user may insure that performance does not degrade when going to the 6.1 release, as a result of the microcoded queue implementation, though he will receive no benefit from it at all.

Six new segments were added to the Micro-MCP to accomodate the data communications facilities in the 6.1 release. The new segments are QUEUE.READ through MESSAGE.COUNT inclusively. Typically, data communication applications which use a handler program generated by the NDL compiler should consider all six

segments to be a part of their working set, though only the first four of the six are concerned with the queue implementation. The MESSAGE.COUNT segment is invoked by the communicate operator of the same name and is used to determine whether or not a message exists in the queues. The DC.INITIATE.IO segment is also invoked by the communicate operator of the same name and should always be considered a part of the working set for any data communications applications.

PROGRAM-DEPENDENT STATIC REQUIREMENTS

The static memory requirements of a program, that memory which is required for everything except the program's code, may be divided into two classes. Three items which are required are fixed in size and the user has no control over them. The user actually has little control over many of the static requirements, though there are some items which he may cause to vary. Items in the latter category are referred to as conditional requirements.

The fixed requirements of the Program Static Memory are composed of three components. These are listed below.

Run Structure Nucleus

This is a table of information constructed by the MCP when the program reaches BOJ. It is a fixed size of 2386 bits.

Interpreter Segment Zero

The size of Segment Zero, the non-overlayable segment, of the Interpreter being used must be determined and added. Space for one memory link must be included.

Interpreter Segment Dictonary

The number of segments in the Interpreter must be determined. The space required for its segment dictionary is then ten bytes times the number of segments plus space for one memory link. (10 X number of segments) + memory link.

The following are the conditional items which must be included in the calculation of Program Dependent Static Requirements.

Program Code Segment Dictionary

The number of code segments which comprise the program may be

determined from the compiler listing of the program. Code segment dictionary space in bytes is then determined by (10 X number of segments) + memory link.

Data Dictionary

The number of data segments used by the program is known to the programmer and is available from the compiler listing. The space for the data dictionary in bytes is calculated by (10 X number of data segments). No memory link is required.

Base-Limit Area (also known as Program Run Structure)

This number is readily available from the compiler listing. It is the total data space required by the program (between Base and Limit Registers). Space for one memory link must be added.

File Dictionary

There is one entry in the file dictionary for each file declared in the program, regardless of whether it is ever used or not. File Dictionary space is given by (10 X number of files declared). No memory link is required.

File Information Block (FIB) Space

This may be calculated in bits by:

1048 x Number of MICR Files open plus
796 x Number of Printer Files open plus
605 x Number of Remote Files open plus
796 x Number of Tape Files open plus
1048 x Number of Disk Files open plus
433 x Number of Queue Files open plus
1048 x Number of all other files open at the time.

FIB Memory Links

One memory link is required for each file that is open.

Total Buffer Space

The number of and the size of the buffer areas associated with each file that is open may be determined from a compiler listing. This size should be totaled and added. If the code file on disk has been modified, however, the size given on the listing may be incorrect. True buffer size may be

determined through an MCP keyboard instruction. (Refer to B1000 Software Operational Guide.)

I/O Descriptors

There is one I/O descriptor, which requires 272 bits of space, for each buffer in each file that is open.

Disk File Headers

Disk file headers are maintained, either in memory or on disk, for all disk files that are open. If the file is processed in a random access mode, the header is maintained in memory. Otherwise, the header is stored on disk and brought into memory when new disk areas are allocated. Each header will require 580 bits plus 36 bits for each area requested by the file declaration, regardless of whether of not the area is allocated, plus space for one memory link. This area is required only when the header is in memory.

Header Dictionaries

Disk file headers are addressed by the MCP through dictionaries. These dictionaries are segmented. One segment contains space for ten dictionary entries. Each dictionary entry is a system descriptor and requires 80 bits of memory. The space required for header dictionaries may be calculated by (800 + memory link) X ((disk files open MOD 10) + 1) bits.

PROGRAM-DEPENDENT DYNAMIC REQUIREMENTS

To determine the working set of segments for any program one must know where a program spends its time or its "main line" of procedure calls. The corresponding segment sizes must then be added up for this main sequence. Segment sizes can be obtained from compiler listings. For RPG programs, all code segments must be included in the working set. For all other programs the compilers produce a list of code segments and sizes. Then the working set segments should be listed and totalled. All segment sizes should have 20 bytes added to account for the size of an associated memory link.

As previously discussed, if any interpreter segments are used by the program, these must also be included in the total.

M-MEMORY MANAGEMENT

The function of M-memory management is to best manage the available control memory (M-memory) in a dynamically changing environment. There are four events which are able to affect the system's demand for M-memory by the introduction or removal of interpreters:

BOJ
EOJ
ROLLIN
ROLLOUT

Upon the occurrence of any of these, if the interpreter set changes, the new demands will be evaluated and M-memory reallocated.

One of two allocation schemes will be employed:

DISTRIBUTION

This method distributes the available M-memory statically among the active interpreters. The size of each portion depends on the interpreter's needs, and the available amount of M-memory. The portion of the interpreter which is not able to be placed in M-memory remains in S-memory. As the number of active interpreters increases, this allocation scheme remains in effect until further dispersion of M-memory would result in a severe performance degradation. When this threshold is reached, the second allocation scheme is put into effect.

CONTENTION

This method dynamically shares M-memory, in the form of n fixed size pages, among greater than n interpreters contending for these pages. When an interpreter succeeds in capturing a page of M-memory, the low-order portion of the interpreter will be copied into the page from S-memory. However, when the page is re-captured by another interpreter, since there is no mechanism for transferring information from M-memory to S-memory, the information in that page will be lost. Hence, all active interpreters must be entirely in S-memory.

DETAILED DESCRIPTION

1. When a new interpreter is to be brought into memory, the

procedure "M.IN.M.OUT" is called. This may be called either from BOJ, EOJ, ROLL.IN, or ROLL.OUT. The last entry in the interpreter dictionary is first stored in "DIC.LAST.LOC". Then the interpreter dictionary is searched for entries whose usercount is equal to zero (thus no longer in use). These entries are deleted by calling "M.CLEAROUT".

The previous allocation method is then stored. If there is no M.MEMORY on the system (B1710 series), then the procedure "NO.M" is called. NO.M examines in turn, each entry in the interpreter dictionary to ascertain if it is in S.MEMORY or not, and if not, the procedure "D.TO.S" is called to bring in the interpreter from disk. The presence bit is then set (although the system has no M.MEMORY), and a pseudo M.MEMORY address is calculated and stored in "ID.M.ADDR". NO.M then exits to M.IN.M.OUT and thence to the procedure which called M.IN.M.OUT.

Assuming that M.MEMORY does exist, the total minimum number of M.MEMORY pages required for all interpreters is added to that required for CSM, then this total number of pages is compared to the total number of pages of M.MEMORY available on the system.

If the total number of pages required is greater than those available, then the contention method is invoked, otherwise the distribution method is invoked. The contention method will be discussed first. For the distribution method, proceed to step 6.

2. The contention method calls the procedure "CNTN.SETUP". CNTN.SETUP first checks to see if the pages remaining after CSM is allocated is less than 2, and if so, then all the interpreters will be contending for the remaining page, including SDL, and the procedure contention is called (proceed to step 3). If the number of remaining pages after allocating CSM is not less than 2, then this number of pages is stored in "M.NUMBER.PAGES". The SDL interpreter is assigned a page, plus any fraction of a page which may be left over. This may occur if CSM does not occupy exactly a full page, normally 1024 words. Next, the number of active interpreters is counted and this number compared against M.NUMBER.PAGES. If M.NUMBER.PAGES is greater than or equal to the number of active interpreters, then the distribution method is called (proceed to step 6). (This could be caused by an interpreter with a very large minimum requirement.)

3. The procedure contention first ascertains if the SDL interpreter is partially resident in S.MEMORY, and either M.NUMBER.PAGES is equal to 1, or the portion of the SDL interpreter in M.MEMORY is greater than the size allocated for SDL. If so, then the procedure "HIL.TO.S" is called,

else proceed to step 4. HIL.TO.S saves the current S.MEMORY address of the SDL interpreter, and stores the disk address of the SDL interpreter in the interpreter dictionary entry for SDL. The procedure "D.TO.S" is then called to bring in the interpreter from disk. D.TO.S looks for memory for the interpreter, makes the found address mod. 16, reads the interpreter into memory and marks the interpreter dictionary entry present. If sufficient memory space was not found, then the previous (partial) SDL interpreter is restored in S.MEMORY, and all procedures exited, returning all zeros to the procedure which called M.IN.M.OUT. Otherwise, the new copy (complete) is marked not present in M.MEMORY and the memory space of the old partial copy marked available. HIL.TO.S now exits, returning to the contention procedure (proceed to step 5).

4. If neither "M.NUMBER.PAGES" is equal to 1 nor the portion of the SDL interpreter in M.MEMORY is greater than that allocated for SDL, and if the portion of the SDL interpreter in M.MEMORY is less than that allowed, then the procedure "LK.OUT.MOR" is called to move more of the SDL interpreter from S.MEMORY to M.MEMORY.

5. The procedure "M.CLEAROUT" is then called to clear out of the interpreter dictionary all partially resident interpreters, with the exception of SDL. Each entry in the interpreter dictionary is then in turn examined, and passed through the procedure "CNTN.LOADR" until all entries are examined, at which time contention is exited to M.IN.M.OUT (proceed to step 10).

   The function of the procedure "CNTN.LOADR" is to load interpreters either from disk to S.MEMORY, and/or from S.MEMORY to M.MEMORY. It first examines the interpreter dictionary entry to determine whether the interpreter is on disk or in S.MEMORY. If it is not in S.MEMORY, then the procedure "D.TO.S" is called to bring the interpreter in from disk. If sufficient memory space is not found, then D.TO.S exits through all procedures, returning all zeros to the procedure which called M.IN.M.OUT. "ID.N.ADDR" and "ID.TOPM" are calculated. Each interpreter is set up for one page of memory. If there is available M.MEMORY left, then the page is overlayed from S.MEMORY to M.MEMORY (proceed to step 10).

6. If the total number of pages required is not greater than those available, then the distribution method is invoked, and the procedure "REDISTRIBUTION" called. The procedure redistribution calculates whether the amount of available M.MEMORY is exactly of a size required to house the minimum requirements of all interpreters and CSM. If so, then the

procedure "M.GRINDER" is called passing a value of 1. (Proceed to step 7).

Otherwise, the total amount of memory required to house the maximum requirements of all interpreters and CSM is calculated and compared against the total amount of M.MEMORY available, and if less than or equal to the amount of M.MEMORY available, then the procedure M.GRINDER is called, passing a value (field WHICH) of zero (proceed to step 7).

If neither of the above conditions is met (that is, neither the minimum nor the maximum of all interpreters will fit in M.MEMORY) then the procedure "DISTRIBUTE" is called, passing a value (field HUH) of zero. The procedure distribute stores the maximum available M.MEMORY, amount required for CSM, then if HUH = 0, it initially assigns each interpreter its minimum required space, increments each one in turn by one page, until all available M.MEMORY is allocated. If HUH = 1, each interpreter's minimum is assumed to be zero, then incremented by one page until all available M.MEMORY is allocated. The procedure M.GRINDER is then called, passing a value (field WHICH) of 2.

7. The main function of M.GRINDER is to reallocate M.MEMORY one of three different ways, depending on the values of "WHICH". M.GRINDER examines each interpreter dictionary entry in turn. After having examined all interpreters, if there is still some M.MEMORY remaining, then proceed to step 9, otherwise proceed to step 10.

If the entry being examined is not in M.MEMORY, or the page being examined is not the current M.MEMORY page, then proceed to step 9. Otherwise, if the size of this page in M.MEMORY is not the size it should be, proceed to step 8.

If this M.MEMORY page is the correct size; and if the interpreter is either partially resident in S.MEMORY or if the total length of the interpreter is less than or equal to the amount of this interpreter currently in M.MEMORY (i.e., the interpreter is entirely in M.MEMORY); and this interpreter is not in S.MEMORY, then proceed to step 9.

Otherwise (that is, the interpreter is entirely resident in S.MEMORY, so the portion of S.MEMORY which was copied to M.MEMORY must be returned), the interpreter is marked as partially resident in S.MEMORY. If the total length of the interpreter is less than or equal to the amount of the interpreter currently in M.MEMORY, then the procedure "ALL.IN.M" is called to return the entire S.MEMORY space for this interpreter. Otherwise, the procedure "LK.OUT.MEM" is called to return the S.MEMORY space corresponding to that portion of the interpreter which has been copied into M.MEMORY.

8. If the amount of the interpreter in M.MEMORY is less than the amount allocated in M.MEMORY for this interpreter, then the procedure "LK.OUT.MOR" is called to copy more of the interpreter from S.MEMORY to M.MEMORY.

9. If this point is reached, then the appropriate interpreter must be brought in from disk.

   The procedure "M.CLEAROUT" is called to clear out all partial interpreters from the interpreter dictionary (with the exception of the SDL interpreter and already fitted interpreters).

   If the current entry in the interpreter dictionary is SDL, then the procedure HIL.TO.S is called (refer to step 3 for the functions of HIL.TO.S). If sufficient memory space is not found in HIL.TO.S, then exit through all procedures passing a value of all zeros to the procedure which called M.IN.M.OUT. Next, each entry in the interpreter dictionary is examined in turn, and if present in S.MEMORY but not in M.MEMORY, then the procedure "S.TO.M" is called to overlay the appropriate page from S.MEMORY to M.MEMORY, and to return either the entire S.MEMORY space occupied by the interpreter or else to return only the portion overlaid. Each entry in the interpreter dictionary is once again examined in turn, and if the presence bit is set, proceed to step 10.

   If the presence bit is not set, then the procedure D.TO.S is called to bring in the interpreter from disk to memory (refer to step 3 for a description of D.TO.S). If sufficient memory is not found in D.TO.S, then all procedures are exited, passing a value of all zeros to the procedure which called M.IN.M.OUT.

   The procedure S.TO.M is then called (see description above).

10. At this point, the allocation method (either distribution or contention) has been decided and executed, and control passed back to M.IN.M.OUT.

    If the new allocation method chosen was successful, and if the new allocation method is the same as the old one, proceed to step 11. If the new method is distribution (therefore, the old was contention), then the procedure RELEASE.A.SEG is called to mark the MCP segment REIN.STATE available (reset save bit in the memory link). If the new method is contention, then the procedure SAVE.A.SEG is called to mark the MCP segment REIN.STATE saved (set save bit in the memory link).

11. If the value passes to M.GRINDER (WHICH) was 0 or 1. Then return from M.GRINDER through redistribution, to M.IN.M.OUT. If the value passed to M.GRINDER (WHICH) was 2, then return from M.GRINDER through DISTRIBUTE to REDISTRIBUTION, to M.IN.M.OUT and thence to the procedure which called M.IN.M.OUT.

PROCESS (PROGRAM) MANAGEMENT

Viewing the MCP as a manager of processes emphasizes its role in the management of job execution. That part of the MCP concerned with such management may be termed the "process controller". While the process controller is not a distinct module in the MCP, it is a convenient term to describe all those distinct functions which, taken together, form a conceptual package. Certain of these functions, namely "ROLLIN", "ROLLOUT", "CAUSE", "HANG PROGRAM", are best understood within this context and will be discussed in depth in this section.

The actual execution of programs, the allocation of processor time to processes which are ready to execute and are, therefore, in the Ready Queue, is accomplished by micro code contained in GISMO known as the "Micro Scheduler". The Micro Scheduler is a part of the process controller. The Micro Scheduler is responsible for the allocation of all processor time on all processors which may be attached to the system.

The process controller is driven by the occurrence of certain software events, called "soft events", which can be identified and anticipated by the MCP. When a process submits a request to the MCP, the process may or may not be required to wait. If a wait is necessary, the MCP is able to anticipate the event upon which that process must wait. Thus the MCP can label the job as waiting for some "soft event", suspend the job by placing it in the "wait queue", and continue to execute its other duties. When the soft event "happens", the Micro MCP can search the wait queue to discover the process marked waiting for the happening of that event.

The "HANG PROGRAM" function, which places programs in the wait queue, and the "CAUSE" function, which takes programs from the wait queue, are crucial. Both functions must be cognizant of the same soft events. "HANG PROGRAM" is responsible for creating a unique bit string which will represent the soft event for a process. On the other end "CAUSE" must have the proper soft event generated for it, so that the waiting process can be located.

The main asset of this method of process manipulation is to free the MCP from waiting for the completion of I/O operations. It is able to initiate a requested operation and to independently match a soft event with its corresponding process at a future time when the operation has been completed.

The process controller receives inputs from two sources: an "I/O DEVICE" or a "CONTROL DEVICE". Both may affect processes in the system. User demands upon the system are submitted through a control device which may accept only control language statements. On the B1000, the supervisory printer (SPO) may only be used as a control device. The card reader may be dynamically assigned as a control device or an I/O device. All other peripheral devices may be used as I/O devices only. In addition, a program may act as a control device by sending a communicate to the MCP which contains a control language statement. See "PROGRAM COMMUNICATES".

Control language statements, of direct interest to the process controller, may be divided into three categories:

(1) Statements which generate a soft event (e.g., allow a suspended process become active, direct a process to a peripheral device)

(2) Statements which cause job suspension

(3) Statements which request job execution and provide all the appropriate parameters

If the control language statement requested that a job be executed, the "Control Language Processor" directs that the job be scheduled. Briefly, the scheduling function involves placing it in the "schedule queue" but allocating no machine resources. In the MCP outer loop, the schedule queue is periodically checked, and the first job in the queue is initialized.

"Program Initialization" involves allocating the machine resources and setting up the structures necessary for program execution. Once the job has been initialized, it is placed in the "READY QUEUE" to await actual execution.

Once a program has been initialized, it will move in and out of six possible states during the course of its life in the system:

READY QUEUE
COMMUNICATE QUEUE
WAIT QUEUE
NOT QUEUED, EXECUTING
NOT QUEUED, COMMUNICATE BEING ANALYZED
M COMMUNICATE QUEUE

The ready queue contains jobs which are ready to run. The communicate queue contains jobs which have requested some MCP function. The wait queue contains jobs which are waiting for the happening of a "soft event".

The queuing mechanism is managed as follows. All run structures are linked in memory by priority. A field in the run structure nucleus, "RS.Q.IDENT", specifies the current state of the process. The first member of a queue can thus be found by searching the linked list of run structures until the proper value in RS.Q.IDENT is found.

A job waiting in the ready queue represents a demand for processor time upon the system. This queue is interrogated by the Micro Scheduler. If a job is found, the reinstate function, which is performed by the Micro Scheduler in GISMO, is called in preparation for turning the processor over to that job. Briefly, the reinstate function performs certain housekeeping duties and causes a processor to begin execution of that job.

The program will execute until one of three things happen:

(1) The program's interpreter discovers an interrupt which requires the MCP's attention.

(2) The program needs some MCP service performed before it can continue

(3) The master processor instructs the slave to idle.

In any case a communicate message is built in a field called RS.COMMUNICATE.MSG.PTR in the program's Run Structure Nucleus and control is passed back to the Micro Scheduler.

The contents of RS.COMMUNICATE.MSG.PTR, analyzed by the "communicate handler" in the Micro Scheduler, specifies what action is to be taken upon the program. In the case of (1) above, the message will simply contain a request to be put back in the ready queue. (The Micro Scheduler then returns to its outer loop where it independently discovers the INTERRUPT.)

A request for service (2) may or may not require that the program wait for the happening of some soft event. If the request can immediately be serviced, the Micro Scheduler does so and places the job back in the ready queue. If the program must wait,

however, the "HANG PROGRAM" function is called.

"HANG PROGRAM" puts the job in the wait queue and labels it as waiting on the appropriate soft event. Depending on the reason for the wait, the program may or may not be "rolled out". The "ROLLOUT" function will copy all but a central core of the program's Run Structure Nucleus to disk. For a detailed discussion of these functions, see their respective sections below.

The program will remain in the wait queue until the event upon which it was waiting has been "caused". The soft event upon which a job must wait may come from three basic sources:

(1) I/O interrupts

(2) Control language statements

(3) MCP

I/O interrupts are "hard events" which must be transformed into soft events before they may be associated with a process. A hard event is any asynchronous occurrence in the hardware of which the software must be cognizant. The occurence of such a hard event is usually manifested by a flag in the processor. The function of "I/O COMPLETE" is to transform those hard events of interest to a process into its corresponding soft event.

Some control language statements will cause the control language processor to generate soft events. Such statements signify the happening of some event a process might be waiting for (e.g., "AX", "IL", "UL", "GO", and "OK").

Other soft events are generated internally by the MCP. For example, processes waiting on a no-memory condition or a parent program waiting for the termination of a nested program must be notified when they are able to resume processing. The MCP generates such soft events.

At the point when the soft event has been generated (from whatever source), one can say that the "event has happened". This soft event is used by the Micro Scheduler function to locate the corresponding process in the wait queue.

If the process is in memory (had not been rolled out), the Micro Scheduler analyzes the reason that the process had been waiting to determine whether or not the last communicate was completed. If it was, the job is put in the ready queue to await reinstatement. If the communicate was not completed, the job is put in the communicate queue to wait for the reinitiation of the communicate by the communicate handler.

If the process was not in memory and memory is available, then the "ROLLIN" function is called. Its duty is to re-establish the run structure that had been "rolled out" to disk. The reason for waiting is then analyzed in the same manner described above. If there was no memory available for roll-in, then the job is put back in the wait queue to wait for memory. The wait reason will be updated to reflect this status change and to specify into which queue the job would have gone had memory been available. When memory becomes available, the job will be put directly into the specified queue.

The process will continue to be manipulated in this fashion until it has completed execution. At that time it will request the end-of-job function from the MCP and terminate.

DEMAND MANAGEMENT

MCP OUTER LOOP

The MCP can be viewed as a program whose sole duty is to respond to demands made upon it by the system. This seemingly innocuous statement is valid even though the MCP is a vastly complex program. The complexity arises, however, by virtue of the diversity of demands to which the MCP is able to respond.

There are five basic categories of demands to which the MCP initially responds. These categories are recognized at the outermost or most global level of the MCP, which iteratively searches for each. Once a demand is found, it is analyzed at increasing levels of detail and resolved according to its specific request. Control is then returned to the outer loop which continues to search for demands.

The five types of demands recognized by the MCP's outer loop are described below.

TIMER INTERRUPT

The first type of demand recognized by the MCP is called a timer interrupt. There are two fields in the MCP's global data space: A software maintained system clock and a clock mask. Every tenth of a second an interrupt is caused by the hardware. GISMO detects this interrupt and bumps the system clock. Every time the MCP begins its loop searching for demands, it checks to see if the value of the system clock has exceeded the value of the clock mask. If it has, the MCP calls the "N.SECOND" routine to perform its housekeeping duties and resets the clock mask to some value greater than the system clock. See "N.SECOND routine".

I/O INTERRUPTS

An I/O interrupt is a soft mechanism by which GISMO notifies the MCP that an I/O operation is complete. GISMO will only do so when the MCP requests that it be notified or when an exception condition has occurred on the I/O operation. This should not be confused with a "service request" type of interrupt. This service request is a hard level in the processor and is used to notify the software that a hardware I/O control is in need of service.

The MCP will request notification of the occurrence of I/O completion only when there is a need for it to know. The MCP does not request the return of I/O complete interrupts on user I/O operations unless the program which caused the operation to be initiated is waiting on the I/O operation. This is discussed further in the sections of the specification covering READ and WRITE.

When an I/O operation is completed, GISMO stores the result descriptor associated with the operation in its proper location in memory. The field is known as the "result descriptor field" and is a part of the actual I/O descriptor. There is an area allocated in memory known as the interrupt stack, which is actually a queue of I/O complete interrupts. GISMO, after storing the result descriptor, if the interrupt request bit in the descriptor was on, stores the address of the result descriptor in the interrupt stack and "causes" the MCP, if it is waiting. In its outer loop, the MCP requests that GISMO deliver the address on the top of the interrupt stack. It analyzes the descriptor at that address and takes the appropriate action. The MCP continues to request addresses from GISMO in this fashion until the stack has been exhausted.

Upon receiving a descriptor's address from GISMO, the MCP invokes a routine called "IO.COMPLETE" to begin the analysis. Depending on the value found in a field of the result descriptor, IO.COMPLETE invokes one of the following MCP facilities, each of which is discussed in depth, on the following pages.

CAUSE MECHANISM (SEE "PROCESS MANAGEMENT")
CONTROL LANGUAGE PROCESSOR
IOAT MAINTENANCE
I/O ERROR HANDLER
SPO MAINTENANCE

JOB SCHEDULING AND INITIALIZAIION

After exhausting the interrupt stack, and if an MCP global, "CHANGE.BIT", is true, the MCP checks the "schedule queue" to determine if any jobs have been scheduled for execution. CHANGE.BIT will be false whenever a previous attempt at program initialization failed because of insufficient memory, and nothing has intervened to create a possibility of success at this attempt. The program initialization routine sets CHANGE.BIT to zero (false) whenever an initialization fails. It is set to one (true) whenever a block of memory is freed by job termination or "rollout", whenever a new job is placed at the top of the active schedule, or whenever explicitly set by the "PS" control language statement. The MCP is thus able to maximize its own resources by

by-passing a futile attempt at job initialization.

The schedule queue contains an active and a waiting schedule. Both are linked lists on disk which contain those jobs awaiting execution but for which no memory resources have yet been allocated. The control language processor identifies a request for a job to be executed. It builds a log entry (see "LOGGING INFORMATION") for that job and links it by priority and time of request to other jobs waiting to be initialized. See "CONTROL LANGUAGE PROCESSOR" for exact specifications. The active schedule lists those jobs that are ready to run. The waiting schedule contains those jobs whose initialization must await the occurrence of some event (i.e., the termination of another job or operator control message). When the event happens, the job is transferred from the waiting schedule to the active schedule, where the MCP will find it.

The MCP selects the first job in the active schedule for initialization. once the job has been de-queued, control is passed to the "program initializer" which allocates the machine resources and sets up the structures necessary for the program's execution.

COMMUNICATES

A program may request certain services from the MCP. These requests represent another class of demands to which the MCP must respond. The "communicate queue" contains jobs which have submitted such a request.

The queuing mechanism is managed as follows. Each run structure nucleus contains two fields: "RS.COMMUNICATE.MSG.PTR" which is a standard message area and "RS.Q.IDENT" which specifies in which queue, if any, the program is. The value of RS.Q.IDENT may be:

```
 0 = READY QUEUE
 1 = COMMUNICATE QUEUE
11 = WAIT QUEUE
-2 = NOT QUEUED (i.e., running)
10 = MMCP COMMUNICATE QUEUE
 3 = EXTERMINATE QUEUE
```

All run structures are linked together by priority. Thus the members of a given queue may be discovered by searching the linked list of run structures and checking RS.Q.IDENT.

The first job in the communicate queue is serviced according to the contents of RS.COMMUNICATE.MSG.PTR. The message is initially analyzed by the communicate message handling routine which calls the proper subroutine to further analyze the message and take the appropriate action. The proper subroutine is determined by the first two bits of this message area called "RS.ITYPE". The values and corresponding meanings of this field are as follows:

00 = INTERPRETER GENERATED COMMUNICATES
01 = PROGRAM GENERATED COMMUNICATES
10 = UNDEFINED
11 = FILE CLEANUP COMMUNICATE

Interpreter generated communicates contain requests from the program's interpreter for services which are unrelated to the program's code. These include requests for missing segments, trace and run time error messages, etc.

Program generated communicates are requests for code related services such as I/O operations. These are specified under "PROGRAM COMMUNICATES".

The file cleanup communicate is an MCP generated communicate used in conjunction with program end-of-job.

PROGRAM REINSTATE

To be specified.

PROGRAM COMMUNICATES

All object programs communicate with the MCP by means of a Communicate S-operator. The operator serves to transfer control from the user's interpreter to the MCP's. Though many communicates are now handled by micro-code in the Micro-MCP, the means of communication has not changed. The compiler generates code which establishes an area in the program's run structure. This area generally conforms to a standard format which is recognizable by the MCP. The fields in this area are defined arbitrarily, however. Only the first twelve bits of the field must conform to the format presented below.

COMMUNICATE FORMAT

VERB	0 - 11
OBJECT	12 - 35
ADVERB	36 - 47
CT.1	48 - 71
CT.2	72 - 95
CT.3	96 - 119
CT.4	120 - 143
CT.5	144 - 167
CT.6	168 - 191
CT.7	192 - 215
CT.8	216 - 239
CT.9	240 - 263
CT.10	264 - 297
CT.11	298 - 321
CT.12	322 - 345
CT.13	346 - 369
CT.14	370 - 393
CT.15	394 - 417

Note: All communicates return a value of @00000000000@ or @0000180000000@ in the RS.REINSTATE.MSG.PTR unless otherwise specified.

All interpreters, when executing the Communicate S-operator, store a pointer to the reserved, formatted memory area in the field called RS.COMMUNICATE.MSG.PTR of the RS.NUCLEUS of the program being executed. This forty-eight bit field specifies not only the relative address of the communicate area, but also the size of the area in bits. For further information on this aspect of the operation, refer to the programmatic description of the Run Structure Nucleus.

If the MCP needs to convey information back to the object program after executing the requested communicate, it does so by setting the field called RS.REINSTATE.MSG.PTR to a selected value. If no information is to be conveyed, this field is set to either @00000000000@ or @0000180000000@ before reinstating the program. Other values, and their associated meanings depend upon the type of communicate being executed, and are described for each communicate in the sections which follow.

```
CT.VERB      00
             ILLEGAL COMMUNICATE
```

READ (MICRO MCP)

```
CT.VERB      01
CT.OBJECT    FILE.NUMBER
CT.ADVERB    BIT
             0      REPORT & RETURN TO USER ON EOF
             1      REPORT & RETURN TO USER ON PARITY
             2      REPORT & RETURN TO USER ON INCOMPLETE I/O
             3      LENGTH ADDRESS PAIR IS PRESENT FOR RESULT MASK FIELD
             4-6    -
             7      STACKERS--STACKER # IS IN CT.3
             8-11   -
CT.1         LOGICAL RECORD BIT LENGTH
CT.2         LOGICAL RECORD BASE RELATIVE BIT ADDRESS
CT.3         RANDOM FILE ACTUAL BINARY DISK KEY
             (RECORD NUMBER INSERTED BY MCP FOR SERIAL FILES)
             OR
             LENGTH OF KEY FOR REMOTE FILES
CT.4         ADDRESS OF KEY FOR REMOTE FILES ONLY
CT.5         LENGTH IN BITS OF RESULT MASK
CT.6         BASE RELATIVE ADDRESS OF RESULT MASK FIELD
REINSTATE.MSG.PTR VALUES
             0    GOOD READ
             1    END OF FILE
             2    I/O ERROR
             3    INCOMPLETE I/O
             4    IMPOSSIBLE SEARCH (RPG SEARCH OP)
```

A READ communicate on the B1000 System serves to deliver a logical record to the user program. It does this by moving the record from the I/O buffer area in memory, where it was previously stored by the CSM, to the user's Run Structure (Base/Limit) area. In almost all cases, the READ, WRITE and SEEK communicates are performed by the Micro-MCP. This has been true since the 5.1 release of the software.

The information passed in the communicate area must include a unique file number. This number is assigned by the compilers and is passed to the MCP in CT.OBJECT. The same statement is true for all communicates which deal with an I/O operation, such as WRITE, SEEK, OPEN, CLOSE, POSITION and so forth.

The communicate information must also contain the base-relative address of the memory area where the record is to be stored and the length, in bits, of this area. These items are passed in CT.2 and CT.1, respectively.

Logical record size is contained in the FPB, which is constructed by the compilers from information contained in the user's file declaration. In the case of variable-length records, logical record size may be contained in the record itself. The length of the user's "work area", contained in CT.1, does not have to be equal to the logical record size. If CT.1 is larger than logical record, size, the movement to the work area will occur left-justified with blank fill on the right. If CT.1 is smaller than logical record size, truncation from the right will occur. in the latter case, information will be lost from the low-order postions of the record.

For a sequential file, the record delivered will be the next record in sequence in the file. For a random disk file, the record to be moved is specified by the binary number in CT.3. Record numbering in a random file on the B1000 system begins with one, by definition, regardless of the source language which the user program is written in. A zero passed in CT.3 will be considered invalid by the MCP and the appropriate action will be taken.

If the user program included code to be performed when the end of the file is reached or, in the case of random files, when CT.3 specifies a number which is beyond the end of the file or describes a record which has not yet been written or is otherwise invalid, the specified bit in CT.ADVERB should be turned on.

If the user program included code to be performed when an I/O error occurs and cannot be corrected by the MCP, the proper bit in CT.ADVERB should be turned on. If control is returned to the user in this case, if the bit in CT.ADVERB is on, the user's work area will contain the record which was read erroneously. In this case, nothing in the work area should be assumed to be valid.

Ordinarily, when a user requests a logical record and the associated physical I/O operation is not yet complete, the program is not allowed to execute until such time as the requested record can be delivered to his work area. For sequential files, I/O operations to fill all of the I/O buffer areas assigned to the file are initiated when the file is opened. The MCP attempts to stay ahead of the user program from that point, initiating a new I/O operation to pre-fill each buffer as soon as it is emptied by the user program.

For some Data Communications applications, it is not feasible for the user program to wait until a requested I/O completes. If, for example, the program is reading cards and the card reader is not ready, it may be a long time until the operation completes. For such programs, the third bit in CT.ADVERB is used. Setting

this bit causes the MCP to return control to the user program, regardless of whether a record was delivered or not. If no record was delivered, the program is informed of the situation by setting proper values in RS.REINSTATE.MSG.PTR. This is discussed in more detail later in this section.

Remote files may consist of more than one data communications terminal. In such a case, it is necessary for the object program to specify the identification of the terminal it wishes to read from. This is accomplished by setting CT.3 and CT.4 to the proper values.

In all three of the cases described previously, where bits one, two or three are set in CT.ADVERB, it is necessary for the MCP to inform the user program of the existing condition. This is accomplished by setting a field in the RS.NUCLEUS to a specific value prior to reinstating the program. The field is defined as RE.REINSTATE.MSG.PTR and is accessed by the user's interpreter as soon as it is reinstated, after doing a communicate. If a valid record was delivered to the user, the message field is set to a value of zero. It will be set to one, two or three if the respective bits are on in CT.ADVERB and the condition assigned to these bits exists.

If a user program READ communicate encounters an end-of-file condition and bit one in CT.ADVERB is not set, the program will be discontinued by the MCP. If a user I/O operation results in an irrecoverable error and bit two is not set in the READ communicate which requests the record, the program will be discontinued by the MCP. If a user program requests the data from an I/O operation which is not yet complete and bit three of the adverb is not set, the program is merely forced to wait for the I/O completion.

For files which are assigned to Data Recorders and other selected card Input/Output devices, the user may specify that the card which was read is to be routed to a certain physical stacker on the device. This is accomplished by setting the specified bit in CT.ADVERB to one and by setting CT.3 to the binary number which designates the physical stacker. In this case, there is never a need for more than one buffer area to be assigned to the file, and the MCP OPEN routine will prevent this from happening. Card I/O operations in this case are not "buffered" and card throughput will decrease accordingly.

For random disk files, a READ communicate may not result in an I/O operation being initiated. If the user who does the READ is the sole user of the file and if the block which contains the requested record is already in memory in one of the user's buffer

areas. the requested record will be simply moved to his work area. This action is not performed if there is more than one user of the file.

WRITE (MICRO MCP)

```
CT.VERB     02
CT.OBJECT   FILE.NUMBER
CT.ADVERB   BIT
            0      REPORT & RETURN TO USER ON EOF
            1      REPORT & RETURN TO USER ON PARITY
            2      REPORT & RETURN TO USER ON INCOMPLETE I/O
            3      LENGTH ADDRESS PAIR IS PRESENT FOR RESULT MASK FIELD
            4-5    -
            6      QUEUE FILES: WRITE TO FRONT OF
                   QUEUE ("STACK").
            7      STACKERS--STACKER # IS IN CT.3
            8-11   PRINTER SPACING  (4 BIT VALUE)
                   0  NO PAPER ADVANCE
                   1  SKIP TO CHANNEL 1  AFTER PRINTING
                   2  SKIP TO CHANNEL 2  AFTER PRINTING
                   3  SKIP TO CHANNEL 3  AFTER PRINTING
                   4  SKIP TO CHANNEL 4  AFTER PRINTING
                   5  SKIP TO CHANNEL 5  AFTER PRINTING
                   6  SKIP TO CHANNEL 6  AFTER PRINTING
                   7  SKIP TO CHANNEL 7  AFTER PRINTING
                   8  SKIP TO CHANNEL 8  AFTER PRINTING
                   9  SKIP TO CHANNEL 9  AFTER PRINTING
                   A  SKIP TO CHANNEL 10 AFTER PRINTING
                   B  SKIP TO CHANNEL 11 AFTER PRINTING
                   C  SKIP TO CHANNEL 12 AFTER PRINTING
                   D  SKIP TO TOP OF FORM (150C LPM PRINTER ONLY)
                   E  SINGLE SPACE AFTER PRINTING
                   F  DOUBLE SPACE AFTER PRINTING
CT.1        LOGICAL RECORD BIT LENGTH
CT.2        LOGICAL RECORD BASE RELATIVE BIT ADDRESS
CT.3        RANDOM FILE ACTUAL BINARY DISK KEY
            (RECORD NUMBER INSERTED BY MCP FOR SERIAL FILES)
            OR
            LENGTH OF KEY FOR REMOTE FILES
CT.4        ADDRESS OF KEY FOR REMOTE FILES ONLY
CT.5        LENGTH IN BITS OF RESULT MASK
CT.6        BASE RELATIVE ADDRESS OF RESULT MASK FIELD

REINSTATE.MSG.PTR VALUES
            0    GOOD WRITE
            1    END OF FILE
            2    I/O ERROR
            3    INCOMPLETE I/O
```

A WRITE communicate on the B1000 system operates in a manner similar to READ. The user program constructs a logical record

somewhere within its Run Structure and communicates with the MCP. The MCP will then move the data from the work area, the address and length of which are described by CT.2 and CT.1 respectively, to the next available I/O buffer area. The program will be allowed to continue as soon as the movement of the data occurs; it is not forced to wait for completion of the actual I/O operation.

As in the case of the READ communicate, either blank-fill or truncation of the record will occur, depending upon the sizes of the work area and the file's logical record. The buffer will be released, which means that the corresponding I/O operation will be initiated, as soon as the buffer area has been filled to capacity. A program is forced to wait for I/O completion if the MCP cannot find an available buffer to which it can move the record. A buffer is unavailable if the previous I/O operation, which may have been initiated some time ago, is not yet complete.

End-of-file is not reported to a user on an output file except in the cases of disk files and some printer files. End-of-file for a disk file is defined to be an attempt by the user to write past the declared size of the file. The declared size of all disk files is maintained in the File Header, a permanent entity created when the file is opened output for the first time.

For files assigned to printers, end-of-file may be defined to be the sensing by the hardware of the physical end of the page. In all cases, this is not actually the end of the page, but rather the sensing of a channel twelve punch in the Carriage Control Tape. This sensing will be reported to user programs, if requested by setting bit one in CT.ADVERB and by setting a bit in the FPB for the file. Notice that, because of the fact that the MCP is examining the result of I/O operations which may have been initiated some time ago, end-of-page is not reported when it occurs, but "n" write operations later, where "n" is the number of buffers assigned to the file.

I/O errors are also reported to user programs on the WRITE communicate, if requested. This information is necessarily of little practical use, on any Burroughs operating system. Ostensibly, the I/O error routines of the MCP(s) should be of such a nature that the need to report this occurrence never arises.

The same Data Communications applications which use bit three of the adverb on READ, use it in a similar manner on WRITE. When using this bit in the adverb, control is returned to the user when a WRITE is requested but the buffer that should be used is not yet available to the MCP. Again, this can be caused by the

device itself going not ready.

Printer spacing information must be passed to the MCP on each WRITE communicate for a file which is assigned to a printer. This is accomplished by setting the proper bits in the adverb, as pictured in the preceding.

Serial disk files may be opened by the user program for both input and output operations. In other words, the file may be opened in such a manner that both a READ and a WRITE communicate are acceptable, with no intervening Close and Open. When this type of OPEN communicate occurs, the MCP will pre-fill all of the buffers, as if the file had been opened INPUT only, but the buffers are released at different points in the READ and WRITE communicate processing.

The MCP will not move buffer pointers at the conclusion of a READ communicate, as it normally does. Instead, it must wait until the next communicate operator associated with that file is received. If the succeeding communicate is a WRITE, it will move the data from the work area to the buffer and change the operation code in the I/O descriptor to a Write. It will mark the program "ready to be reinstated", and then rotate the buffers in anticipation of the next communicate operator. In this specification, the term "rotate the buffers" means that the MCP moves the necessary buffer pointers and initiates the I/O if necessary.

If the next communicate received at this point is a WRITE, the MCP, after insuring that the next buffer is available for use, will move the data again, from the work area to the buffer and rotate the buffer pointers. If the communicate had been a READ, the MCP would have moved the data in the opposite direction and it would not have rotated the buffer pointers.

In summary, for this type of file, two successive READ operations will move two successive records from the file to the user's work area. Two successive WRITE operations will cause two successive records to be written into the file. A sequence of operations such as READ-WRITE-READ-WRITE will cause two successive records to be delivered to the user and the same records, but not necessarily the same data, to be written in the file. The End-of-File pointer for a sequential file may be extended when the file is opened in this manner.

Disk files which contain variable-length records may not be opened for both input and output operations, or for random access processing.

For Sequential I/O files, a physical I/O operation is not necessarily initiated each time the user program does a WRITE. For blocked files, if the user has done a WRITE on any record in the block, the operation will be initiated only when the buffer pointers are moved past the end of the block.

For data communications files, the fields described as CT.3 and CT.4 are used on WRITE communicates exactly as they are on READ communicates.

For random disk files a WRITE communicate may result in more than one physical I/O operation. If the file is blocked, the block which contains the requested record must be in a buffer in memory before the record is inserted in the block and actually written to disk. This is due to the fact that the hardware can only initiate I/O operations and terminate them on segment boundaries.

If the block which contains the requested record is not in memory when the WRITE is issued, the MCP will initiate a Read operation, force the requesting user to wait for its completion, move the record into its respective position in the block after the I/O completes, allow the user to be reinstated at this point and initiate the requested Write operation, if the file is being accessed in the RANDOM mode.

In the 6.1 release of the software, a file access method known as DELAYED RANDOM was implemented. When DELAYED RANDOM is used, the first request for a logical record of a given block of a DELAYED RANDOM file will result in a physical I/O which reads the necessary block into memory. Subsequent accesses to the block will not generate any physical I/O's as long as the block remains in memory. A block is overlayed if a request is made for a block not currently in memory, at this time the least recently accessed block is chosen as the one to overlay. If the chosen block has been updated in memory it is written to disk before the new block is read. Periodically, all blocks that have been updated in memory are written to disk by the SMCP.

SEEK (MICRO MCP)

CT.VERB	03
CT.OBJECT	FILE.NUMBER
CT.ADVERB	-
CT.1	-
CT.2	-
CT.3	RANDOM FILE ACTUAL BINARY DISK KEY

The SEEK communicate is an instruction to the MCP to position the arms properly, on movable-arm devices, and to fill one of the buffers assigned to the file with the block of data which contains the requested logical record. This communicate is applicable to random disk files only. The user is not forced to wait for the completion of an I/O operation initiated by a SEEK communicate. He may be forced to wait if there is no buffer available to use for the operation.

The SEEK communicate may be used by the user programmer to mask some or all of the time required by a READ communicate with computation. It may also be used, prior to a WRITE communicate, to eliminate the necessity of waiting for a buffer to be pre-filled when using blocked files.

No data is moved to or from the user's work area by the logic of a SEEK communicate.

SORTER CONTROL

CT.VERB	04	
CT.OBJECT	FILE.NUMBER	
CT.ADVERB	BIT	
	0-4	-
	5	TRANSFER
	6	POCKET SELECT
	7	STOP-FLOW
	8	BATCH-COUNT
	9	POCKET LIGHT
	10	-
	11	ENDORSE
CT.1	POCKET NUMBER	
CT.2	BASE RELATIVE TRANSFER ADDRESS	
CT.3	BIT LENGTH OF TRANSFERRED DATA	

The SORTER CONTROL communicate is used in conjunction with files assigned to Reader-Sorters only. Such files may be utilized properly in COBOL programs only. Other languages may include portions of the syntax necessary for proper use of a Reader-Sorter, though only COBOL contains everything that is necessary.

When the MCP receives an I/O Complete interrupt from the Reader-Sorter, it immediately references the program which is using that sorter, determines the memory address of the "USE ROUTINE work area", and places a formatted copy of the result descriptor from the I/O operation followed by an image of the item itself in the work area. It then reinstates the user at the code address of his USE ROUTINE. (For additional information on

Item Processing, refer to the B1000 COBOL Reference Manual, Form Number 1057197.)

The MCP takes the action described above regardless of other processing that is occurring. The action described is commonly known as "High-Priority Interrupt Handling".

Only three of the five possible adverb bits may be set in a communicate addressed to the MCP while the user program is executing the USE ROUTINE. These three bits are TRANSFER, STOP-FLOW and POCKET SELECT. The TRANSFER bit is discussed in a subsequent paragraph. If the POCKET SELECT bit is set, the MCP will use the value in CT.1 as the pocket number on the sorter for that item. If the STOP-FLOW bit is set in the adverb, the MCP will also issue the appropriate I/O Descriptor to the sorter. After receiving the communicate, regardless of the adverb bits, the MCP will continue doing whatever it was doing at the time the interrupt was received; the user must give up control at this point.

Pocket selection on the sorter thus happens asynchronously with everything that is occurring on the system, except the sorter. This is currently the only device connected to the B1000 which operates in such a manner. The necessity for this action is dictated by the fact that the sorter is actually a "real-time" device and must be serviced in a specific time period after a check has been read by the hardware.

The TRANSFER bit and its function was added to the 8.0 version of the MCP. When the TRANSFER bit is not set, which will be the case for all programs compiled prior to the 8.0 release of the software, the MCP, upon receiving the POCKET SELECT communicate, will dispatch the pocket number supplied to the sorter control and place an image of the item in a "tank" area in memory. The number of items that may be contained in the tank area is specified by the user and corresponds to the number of buffers requested for the sorter file. In actuality, there will be only one buffer and I/O descriptor, regardless of the number requested, but the buffers requested will be used to determine the size of the tank area.

Item images will be removed from the tank when the user program does a SORTER READ operation on the sorter file. The images will be delivered in sequence to the program. Obviously, the tank area will become full if items are introduced to the system more rapidly than the user program does SORTER READ operations. If this occurs, the MCP will dispatch a STOP-FLOW I/O descriptor to the sorter control, thus stopping the introduction of items. Flow will be automatically started by the MCP when the tank area

is again empty. In this manner, the system prevents TOO_LATE_TO_POCKET_SELECT and TOO_LATE_TO_READ conditions from occurring.

If the TRANSFER bit is set in the SORTER CONTROL communicate, the MCP will not tank the actual image of the item but will store the data at the location specified by CT.2 and CT.3 from the program's run structure. In this manner, the user may cause the MCP to tank whatever he chooses, thus eliminating the need for several programming steps from the user program. A maximum of one hundred characters may be passed and tanked per item.

The BATCH-COUNT bit in the adverb is used to advance the batch counter on the sorter by one, each time it is received by the MCP. This adverb bit will only be accepted by the MCP when the user program is not in the POCKET SELECT USE ROUTINE, and a High Priority Interrupt condition does not exist.

Each pocket on a Reader-Sorter has a red indicator lamp, visible to the operator, above it. The lights may be turned on programatically by the object program issuing a SORTER CONTROL communicate with the POCKET LIGHT bit in the adverb set. Upon receiving such a communicate, the MCP will issue an I/O descriptor to the sorter which will instruct it it turn on the light above the pocket specified by CT.1. The hardware will only take such action when the flow of items through the sorter has been stopped. The same is true of the BATCH COUNT operation.

SORTER READ (MICRO MCP)

CT.VERB	05
CT.OBJECT	FILE.NUMBER
CT.ADVERB	-
CT.1	READ AREA BIT LENGTH
CT.2	READ AREA BASE RELATIVE BIT ADDRESS

Check (item) images are passed to the user program asynchronously. As described above, an item image is passed to the program whenever one is available to the system. The user program is expecting to READ these images synchronously, however, by issuing SORTER READ communicates.

The MCP therefore temporarily stores these images in memory, passing them to the user program in succession, upon receiving this communicate. (Notice that the user program has already seen the images in his POCKET SELECT USE ROUTINE.) This operation is commonly known as "tanking".

The operation of the SORTER READ communicate is similar to that of READ. Item images are passed to the user's work area by the MCP; the length and location of the work area is specified by CT.1 and CT.2 respectively.

There is actually a secondary purpose to the SORTER READ communicate; it informs the MCP fo the user program's processing rate. As described above, images are passed immediately to the user for pocket selection but any other communicate from within a POCKET SELECT USE ROUTINE is prohibited. The images may not be written to disk or saved by the user in any manner, except when they are received via a SORTER READ communicate.

Therefore, if the soft "tanks" of item images maintained by the MCP begin to fill up, which indicates that the sorter is delivering images faster than the user can process them, the MCP will automatically stop flow on the sorter until the user program catches up. The sorter may therefore operate sporadically, in bursts, but all items will at least be pocket selected.

The image of the item in the tank is preceded by a twenty-four digit (ninety-six bit) expansion of the actual result descriptor received from the hardware in connection with that item. This is passed to the user program on the SORTER READ communicate, just as it is placed in his USE ROUTINE work area prior to reinstating his USE ROUTINE.

Though only two communicate formats are implemented for use with Reader-Sorters, the MCP must do a lot more to make this operation possible. A program which opens a sorter causes many different items to be marked non-overlayable in memory. This is described more fully under the OPEN communicate. For a more comprehensive explanation of Reader-Sorter operation, refer to the B1000 COBOL Reference Manual, Form Number 1057197.

OPEN (DM)

CT.VERB	06	
CT.OBJECT	INVOKE NUMBER & PATH NUMBER	
CT.ADVERB	BIT	
	0	INCLUDES PACKID OF DICTIONARY
	1	-
	2	DM.STATUS FORMAT
		0=BINARY
		1=4-BIT DECIMAL
	3	ON EXCEPTION
	4	UPDATE
	5	REORGANIZATION (REORG ONLY)
	6-11	-

```
CT.1        DM.STATUS REGISTER BIT LENGTH
CT.2        DM.STATUS REGISTER BASE RELATIVE BIT ADDRESS
CT.3        DATA BASE NAME BASE RELATIVE BIT ADDRESS
CT.4        DATA BASE NAME BIT LENGTH
CT.5        PACKID BASE RELATIVE BIT ADDRESS (BIT 0 OF
              CT.ADVERB = 1)
CT.6        PACKID BIT LENGTH (BIT 0 OF CT.ADVERB = 1)
```

CLOSE (DM)

```
CT.VERB     07
CT.OBJECT   -
CT.ADVERB   BIT
            0-1   -
            2     DM.STATUS FORMAT
                  0=BINARY
                  1=4-BIT DECIMAL
            3     ON EXCEPTION
            4-11  -
CT.1        DM.STATUS REGISTER BIT LENGTH
CT.2        DM.STATUS REGISTER BASE RELATIVE BIT ADDRESS
```

OPEN

```
CT.VERB     08
CT.OBJECT   FILE.NUMBER
CT.ADVERB   BIT
            0     INPUT
            1     OUTPUT
            2     NEW FILE
            3     PUNCH
            4     PRINT
            5     NO REWIND/INTERPRET (DATA RECORDERS)
            6     REVERSE/POCKET (CARD PUNCH)
            7     LOCK
            8     LOCKOUT
            9     REPORT FILE MISSING
           10     REPORT FILE LOCKED
           11     OVERRIDE NAMING CONVENTION AND SECURITY
REINSTATE.MSG.PTR VALUES
            0    GOOD OPEN
            1     FILE NOT PRESENT (INPUT DISK)
                  PACK NOT PRESENT (OUTPUT DISK)
                  NO MORE FILES ON MULTI-FILE REEL (TAPE)
            2    FILE LOCKED  (DISK FILES ONLY)
```

The OPEN communicate serves primarily to associate a physical file with the logical file declared in the user's program. The communicate has other functions and is also used when such an association has already been made. Basically, the processing invoked by an OPEN communicate obeys the rules set forth in the

definition of the COBOL language.

The object program must pass the unique file number assigned to the file by the compiler in CT.OBJECT. The MCP will use this number to obtain the disk address of the FPB constructed by the compiler for that file. It will read the FPB into memory, allocate memory to contain the FIB, the proper number of I/O descriptors and the buffer areas for the file. It will then construct the FIB, based upon the information in the FPB, the physical characteristics of the device assigned to the file and, in some cases, the logical characteristics of an existing file.

The memory area allocated for a file is, except in the case of a Data Management file, a contiguous area. One memory link only is necessary to describe a file area. The file area will contain all of the items mentioned in the preceding paragraph. FIBs vary in size, depending on the type of device assigned. No memory is allocated for this purpose until a device assignment has been made.

One of the first tests made in the OPEN routine is, "Is the file already Open?". This is a violation of the rules of all languages and the MCP has no choice, if the test is true, but to discontinue the program. There cannot be two consecutive OPEN communicates on the same file without an intervening CLOSE communicate.

Another preliminary test is , "Has a device assignment already been made?". If true, the OPEN processing follows a different course. Device assignment is of prime importance to the OPEN routine.

Device Assignment (Except Disk)

A third preliminary test is whether or not the file is to be assigned to disk. If the file is a disk file, the course of action followed is described under the heading "Disk File Assignment". The remainder of the discussion under Device Assignment applies to non-disk files, that are being Opened for the first time.

The next major test made by the OPEN routine is whether the file is being opened for input, output or both. Only certain card devices, such as Data Recorders, may be opened for both input and output, exclusive of disk files. Certain other combinations of the various bits in the adverb are also illegal. These will be discussed in turn. For the case mentioned above, attempting to

open a card reader, for example, for output purposes will result in the program being DS-ed by the MCP.

If the file is being opened input, the MCP will attempt to match the external names in the FPB of the file, FPB.MULTI.FILE.ID and FPB.FILE.ID, with the labels read previously by the STATUS routine on each peripheral device. If no match is found, the operator is notified and the program is forced to wait until a file with the requested label is introduced to the system, or until the operator resolves the "No File" condition is some other manner. System SPO and Control Card syntax is available to allow the operator this alternative. The program will removed from memory if possible.

If a match is found on two or more units, the operator is notified of this also and again, the program is forced to wait. The MCP cannot recover automatically from this condition; the operator must inform it that he has resolved the "Duplicate File" situation. Again, system SPO and Control Card syntax is available to do this.

The MCP's Control Card routine is invoked whenever a card input device goes from a Not Ready condition to a Ready condition. The routine then reads the first card from the device. If this card, or in some cases, a subsequent card causes a job to be scheduled for execution, the Control Card routine retains control of the MCP, reads the next card, and processes it. It will continue to retain control until the card reader goes not ready, or until a DATA card is encountered. If the Control Card routine terminates processing due to the encountering of a DATA card, the physical input file described by the DATA card is associated with the last job which it placed in the schedule. This is only true if the Control Card routine did not lose control between the time it encountered the card which caused a job to be scheduled and the time it encountered the DATA card.

The MCP will not report a Duplicate File situation, if one exists, if the input file being opened is a card file or a pseudo-reader file and if the job has a physical file associated with it in the manner described above. Rather, the MCP merely allows the associated physical file to be opened by the job, provided the external identifiers in the physical label and in the FPB are equal.

Control Card syntax is provided to allow the operator to specify the physical unit which contains a specific logical file. This specification may be made when the job is scheduled for execution or it may be made permanently by modifying the FPB in the program's code file on disk. If such a specification has been

made, the MCP's OPEN routine will not attempt to match external identifiers, but will simply assign the physical file on the requested unit to the logical file being opened, provided, of course, that the unit is available for such an assignment. It should be noted that making such a specification in an FPB also changes the hardware type in the FPB to match the hardware type of the unit specified. Units are specified by mnemonic name.

If the unit being opened is a tape unit, additional tests are necessary before the device may be assigned. The Reel Number field in the unit's label must match the corresponding field in the logical file's FPB. For tape units, Multi-File Identifiers, File Identifiers and Reel Numbers must all be equal. Also, in the case of a tape file, Control Card syntax is provided to allow the operator to specify the Serial Number of a particular reel of tape. If this is done, all four conditions must be met. As in the case of unit mnemonic specification, Serial Number specification may be made when the job is scheduled for execution or it may be made permanently.

The MCP will allow tape files to be opened when the user programmer does not know the logical record and physical block sizes actually written on the tape. These fields are left unspecified in the FPB by the compiler but the default bit in the FPB, FPB.DEFAULT, is turned on. The recording mode of a tape file may also be left unspecified if the bit is set. The MCP will insert the proper values into these fields when the tape is opened, provided the information is present in the tape's label. If the information is not present, the program will be discontinued when the OPEN is attempted.

The MCP will also insert values for record and block sizes when all card input files are opened and FPB.DEFAULT is set.

The MCP will discontinue any program which attempts to open a file contained on seven-track tape if the logical record size contained in the FPB for the file is not modulo six and if the programmer has specified the tape to be read without hardware translation to EBCDIC. This is true regardless of which bit, Input or Output, is set in the Adverb.

When the MCP receives a request to OPEN a file for output purposes, one of the first items that must be checked is whether or not the file should be assigned to a Backup device, which may be tape or disk. If the user has requested that the file be directed to backup, this will occur before the search for a suitable output unit is made. Backup capabilities are discussed in a separate part of this document.

Assuming that the file is not to be sent to Backup, the MCP next checks to see if the user programmer has requested that the file have special forms for output. This test is made for all output files, regardless of device type. If FPB.FORMS is set to one, which indicates that special forms are required, the MCP will print an appropriate message and force the program to wait until the operator replies in the proper manner. Syntax is provided to allow this. When the operator replies, the OPEN routine is again invoked and the file will be assigned to the device specified by the operator, is any, provided the unit available.

In the absence of a Forms specification by the user, the MCP will search the IOAT for an available unit of the type specified by FPB.HDWR. As described in a prior section, certain values which this field may contain are not actually hardware types but specify a group of types, such as "any tape", "any head-per-track disk" and so forth. In order to be available for output purposes, the unit must be ready, must not be currently in use by another program and must be write enabled.

If the MCP cannot find an available unit of the type requested, it will check to see if it is permissible to direct the output to a Backup device. If so, it will attempt to do so. This is also discussed in the section of this specification describing the Backup operation.

If there is no available unit of the type specified by the user and if it is not permissible to direct the file to a Backup device, the MCP will print a message on the SPO to inform the operator that such a unit is required before the program, which will be identified, can proceed. The program will be forced to wait at this point, and will be removed from memory, if possible.

The MCP may recover the program from this condition automatically, with no operator intervention. If a suitable unit becomes available for output purposes, the OPEN communicate processing for the program will be repeated. Control Card and Keyboard syntax is provided to allow the operator to override the hardware type specified in the user's FPB. Syntax is also provided to allow the operator to force the OPEN processing to be repeated. In this case, the MCP merely tries again.

The MCP will automatically discontinue any program which attempts to OPEN, for output purposes, a file whose hardware type specifies a paper tape reader, a reader-sorter, a card reader, a System SPO or an unknown device.

Certain devices on the B1000 may be opened for both input and output operations. These devices are all card devices; no tape unit may be opened for both types of operations, except via the Emulator Tape constructs. At the present time, there are only three such devices, and they have come to be commonly known as "Data Recorders". Actually, according to the B1000 Systems Index, P.S. 1904 5681, they are the:

1. B9418-2 80-Column Keypunch-Printer
2. B9419-2 96-Column Keypunch-Printer
3. B9419-6 96-Column Keypunch-Printer-Sorter

All of the devices in the above list have one "Wait Station". A Wait Station is used for holding the physical card after it has been read, or at least fed from the input hopper, and before it is printed or punched or both. All of the devices listed have at least one hardware buffer, capable of holding the information contained on one card. This buffer is used on input operations only. "Input" as used here will mean input to the computer.

The table below present the number of input hoppers and output stackers on each physical device.

Device	Hoppers (Input)	Stackers (Output)
B9418-2	2	2
B9419-2	2	2
B9419-6	2	6

Three different I/O Controls are used to interface the devices to the B1000 system. They are:

1. MFC-1 (P.S. 2208 3034)
2. MFC-2 (P.S. 2208 3034)
3. CRPC (P.S. 2211 1371)

The following I/O operations are defined in the appropriate product specification for all of the controls.

READ - (From the buffer in the peripheral). This operation is known, for conversational purposes, as the REPEAT.READ. Valid variants are:

Stacker Select
Inhibit Feed
Hopper Select

STACKER.SELECT.AND.READ - (Read the information from the next card in the hopper). Valid variants are:
Stacker Select
Inhibit Feed
Hopper Select

PUNCH. Valid variants are:
Stacker Select
Inhibit Feed
Hopper Select

PRINT. Valid variants are:
Stacker Select
Inhibit Feed
Hopper Select

PUNCH-PRINT. Valid variants are:
Stacker Select
Unequal Data
Inhibit Feed
Hopper Select

PUNCH-PRINT.AND.READ Valid variants are:
Stacker Select
Unequal Data
Hopper Select

The Stacker Select variant is actually a three-bit field in the I/O Descriptor. When the three bits are set to numeric values other than zero and seven, the device routes the card to the stacker selected by the descriptor. Valid numeric values for the devices range from one to six. The MCP does not, and cannot, edit the numeric value passed by the object program to ascertain that it is valid for the connected device.

The Unequal Data variant is valid only in I/O descriptors which cause a Punch-Print operation to be performed. If the variant is not set, the device will print the same information that is punched on the card. In other words, the beginning memory address for the information to be printed is the same as the beginning memory address for the information to be punched. If the variant is set, the information to be printed will be taken from a memory location which immediately follows the information that is punched.

The Inhibit Feed variant causes the Wait Station in the device to be empty at the completion of the operation. No cards will be

fed from the input hoppers if this variant is set in the descriptor.

The Hopper Select variant causes the feed card to be taken from the secondary input hopper, if there is one. If the Inhibit Feed variant is set, the Hopper Select variant is ignored.

There is a limitation, which was mentioned briefly in a prior paragraph. The MCP cannot distinguish between the B9419-2, which has two output stackers, and the B9419-6, which has six. The MCP does not edit stacker numbers before it sends them to the control. Therefore, programs which utilize all six stackers on the B9419-6 may not be transported to systems which do not have such a device. Even if the MCP could distinguish between the two devices, the editing would have to be micro-coded and would seriously degrade performance.

The capabilities available to the object programmer are programmatically selected by variants on the OPEN communicate and by variants on the READ and WRITE communicates. The discussion is categorized according to the different types of OPEN available.

When the MCP receives an OPEN request with none of the bits in the ADVERB area set, it will assume that the program only wants the information contained on the cards and does not intend to do stacker selection, Read-Punch operations, or any other variation available in the hardware. The I/O descriptors constructed as a result of this type of OPEN will all be of the Repeat.Read type. There may be any number of buffers associated with the file. All of the buffers will be filled when the file is opened. Stacker Selection is not allowed when the file is opened in this manner.

When the MCP receives an OPEN request with the Pocket bit set, it will construct one descriptor, the operator field of which will contain a STACKER.SELECT.AND.READ instruction. The buffer will not be filled when the file is opened. The first READ on the file will cause the first card in the data deck to be moved past the read head and stopped in the Wait Station. The data from the card will be transferred to the object program's work area before control is returned to the program. Stacker Select information passed on the first read may or may not be passed on to the device and will have no effect at all.

The second and all subsequent reads should have Stacker Selection information associated with them. The MCP will not, however, include code to insure this. The MCP will not allow more than one buffer to be associated with this type of file.

This action is distinctly different from the most common type of READ request handled by the MCP. The actual operation is always issued after the communicate is received, and never before, as it is with almost all types of input files. The I/O operation can never be comleted ahead of the demand for it. This operation is, however, similar to the current MCP action for Sequential I/O files on Disk. The file can actually be thought of as an Input/Output file, for all practical purposes. It must be considered such a file by the MCP, in order for the I/O to be initiated at the proper time in a card read cycle. The OUTPUT bit in the OPEN adverb should never be set when the OPEN is requested, however. This will cause the communicate to have an entirely different meaning.

Quite obviously, due to the differences in timing, it is mandatory that the object program close and re-open the file in order to change from pure INPUT to INPUT WITH STACKERS, and conversely.

A number of variations are possible when the device is opened as an output file. There variations are:

1. PUNCH
2. PRINT
3. INTERPRET
4. POCKET

The Pocket variant may be applied to any of the first four variations, or it may be the only adverb associated with the OPEN statement. The MCP, upon receiving an OPEN communicate without PUNCH, PRINT or INTERPRET requested, will assume that Punch is desired. Therefore, OPEN OUTPUT is equivalent to OPEN OUTPUT WITH PUNCH and OPEN OUTPUT WITH STACKERS is equivalent to OPEN OUTPUT WITH PUNCH, STACKERS.

OPEN OUTPUT WITH PRINT means that the program does not want to punch anything into the cards. It only wants to print information.

OPEN OUTPUT WITH PUNCH, PRINT means that the program wants to punch information into the cards and wants to print different information on them. The MCP will allocate the number of buffers requested by the program; each buffer will be 192 bytes long. The MCP will expect to receive 192 bytes of information on each WRITE communicate, 96 of which will be punched in the card and 96 of which will be printed. As always, it is not mandatory that

the program deliver the full 192 bytes. The move from the work area to the buffer is left-justified with blank fill.

OPEN OUTPUT WITH INTERPRET means that the program wants to punch 96 bytes of information into the cards and print the same data. The MCP will allocate the number of buffers requested; which must be at least two, each will be 96 bytes in length. The I/O descriptor constructed will specify a PUNCH-PRINT operation, and the UNEQUAL DATA bit will be set to zero. The INTERPRET request will have precedence over PUNCH, PRINT and a combination of the two. OPEN OUTPUT WITH PUNCH, PRINT, INTERPRET should probably be rejected as a syntax error by the compilers. The MCP will accept the communicate, however, and assume that the programmer meant only INTERPRET. The same applies to OPEN OUTPUT WITH PUNCH, INTERPRET and OPEN OUTPUT WITH PRINT, INTERPRET.

The POCKET variant may be specified on any valid request for an OPEN OUTPUT. The variant is ignored by the MCP on the OPEN request. This is not true for OPEN INPUT WITH STACKERS. It must be true for OPEN OUTPUT, however, to avoid problems which may arise when the device assigned to the file is changed by a FILE card or an "OU" message. The WRITE communicate contains a bit which requests stacker selection. The MCP examines this bit and takes appropriate action on each WRITE communicate.

All of the variations possible when a file is opened OUTPUT are also possible when the file is opened INPUT.OUTPUT When the MCP receives an OPEN INPUT.OUTPUT request with none of the variants set, it assumes that the user wants to read the information from the cards, and punch additional information into them, and print the same information on them. Therefore, OPEN INPUT.OUTPUT is equivalent to OPEN INPUT.OUTPUT WITH INTERPRET.

The adverbs PUNCH and PRINT are relatively useless when the file is opened INPUT.OUTPUT. Both punching and printing occur when the PUNCH-PRINT.AND.READ I/O operator is dispatched to the control. There is no way the device can be made to print and read or punch and read only. These opeations can be simulated, of course, by setting the UNEQUAL.DATA bit in the descriptor and loading the proper portion of the buffer with blanks. This requires some action on the part of the programmer.

The UNEQUAL.DATA bit is set in the I/O descriptor when the MCP receives an OPEN communicate with both the PUNCH and PRINT bits set in the adverb. As in the case of OPEN OUTPUT, the INTERPRET bit has precedence over both PUNCH and PRINT. OPEN INPUT.OUTPUT WITH PUNCH, PRINT, INTERPRET should be considered a syntax error by the compilers. When the MCP receives such a request, however, it generates a PUNCH-PRINT.AND.READ I/O descriptor with

the UNEQUAL.DATA variant reset.

Files opened with various attributes require or can only use various number of buffers.

Attributes	Requires	Can Use
Input only, not Stackers	1	infinite
Input only with Stackers	1	1
Output only	2	infinite
Output and Input	3	3

If the number of buffers specified for a file is less than the number required, it will be allocated the number required. If the number specified is more than the number that can be effectively used, it will be allocated only the number it can use.

As in the case of OPEN INPUT WITH STACKERS, the MCP will not fill the buffers when the file is opened. The first READ issued for the file will cause a card to be fed, read and stopped in the wait station. The information from the card will be passed to the object program at that point. It may be necessary for the MCP to treat the first READ on such a file differently from all other reads. This should be of no concern to either the compiler or the object programmer.

After the first READ on the file, the MCP will normally expect to receive two communicates for each card passed through the device. There shold be one WRITE request and one READ request for each physical card. The program should pass 96 bytes, or 192 bytes, of information to the MCP on each WRITE request. The information will be moved from the program's work area to the buffer on the WRITE communicate. The actual I/O operation will be initiated at this time and the program will be allowed to continue, without waiting for the completion of the operation.

The MCP will normally expect to receive a READ communicate at this point, and the program may be forced to wait for the completion of the I/O operation issued previously. After completion, the information read from the card will be passed on the READ communicate, as always.

It is not mandatory that the program always follow the READ/WRITE sequence described in the foregoing. If the program issues two successive WRITE requests, the input information on one of the

cards involved will be lost to the program. The consequences if a program issues two successive READ requests are somewhat more dire. The information punched and printed on the first card will also be punched and printed on the second. Though this sounds rather bad, this could possibly be of some use to someone.

Since the actual I/O operation for this type of OPEN is initiated on the WRITE communicate, any stacker selection information must be passed along with the WRITE communicate. Stacker information passed on the READ will be ignored.

The MCP will automatically discontinue programs that attempt to OPEN a file on any of these devices if:

1. Neither the Input bit nor the Output bit is on in the Adverb,
2. Both the Print and the Interpret bits are on in the adverb, or,
3. The program is attempting to use a 96-column device in the binary recording mode.

Disk File OPEN

When a user program attempts to OPEN a file which is assigned to disk, the first test made in the Open Routine is whether the file is a new file which the program is creating for the first time or an old file which already exists in the disk directory. In the first case, a disk file header will eventually be constructed in memory by the OPEN routine. In the second case, the disk file header already exists and is stored in the directory and will have to be brought into memory by the Routine. The Open procedure for a new file will be discussed first.

Programs which attempt to open new files for input and output in the serial access mode will be automatically discontinued at this point. Also, programs which attempt to open a new Multi-Pack File and have blanks in the PACK.ID field or the FPB will be automatically discontinued.

If the Open communicate specifies that the file is a code file, the proper names for the file will be stored in the FPB at this point. Also, the number of disk areas requested in the FPB will be automatically set to one. The fact that this is a code file will be recorded by the MCP in the FPB for the file. This information is required when the file is closed.

If the PACK.ID field of the FPB contains something other than EBCDIC blanks, the program is requesting that the file be directed to a user pack with that ID. The MCP will, at this point, examine the Pack Information Table maintained in memory for a pack with the corresponding identifier. If such a pack is present on the system, the routine continues. Otherwise, if the user has requested that he be notified when the pack is not present by setting the REPORT FILE MISSING bit in the adverb, he will be so notified at this point and control will be returned to the user through the normal processor queue mechanisms.

If the REPORT FILE MISSING bit is not set, a message to the operator will be displayed and the program will be suspended until the requested pack is introduced to the system or the operaor overrides the PACK.ID specified. Control syntax is provided to allow the operator several means of accomplishing this.

If the file being opened is a multipack file and if the serial number of the physical pack is zero or if the pack is already a continuation pack for another multipack file, the program will be automatically discontinued. If the number of areas requested for the file by the programmer is greater than 105, it will be automatically set to 105 by the MCP. In the latter case, no warning is sent to the operator.

Memory for the file header is allocated at this point. If the user had requested that the disk areas to be assigned to the file be allocated when the file is opened, the allocation is done at this point, provided sufficient disk is available. If sufficient disk is not available, the program is suspended and an appropriate message is displayed on the SPO.

The File Header is now constructed in memory, based upon information contained in the FPB. The ultimate dispensation of the header is dependent upon the type of CLOSE communicate performed on the file.

At this point in the OPEN processing, the logic becomes the same for new and old files. Before proceeding with a description of the logic at this point, it will be advantageous to describe the processing which occurs when the user opens an existing file.

When the user requests an Open of an existing file, the first occurrence is a determination of whether or not the file is present on the system. All three names in the FPB must match an

existing file if the name fields contain something other than EBCDIC blanks.

If the PACK.ID Field is blanks the file is assumed to reside on system disk. The directory on the system disk will be searched. If the PACK.ID field is not blanks, it specifies that the file exists on a removable user pack of that name. If there is no such pack on the system at that time, the program is suspended, with an appropriate operator message, until the pack is introduced to the system or the operator overrides the PACK.ID in the FPB. There are several means provided for the operator to accomplish this. If the REPORT.FILE.MISSING bit is set in the communicate adverb, the program is not suspended, but control is returned to it through the normal processor queues and the fact that the pack is not present is reported to it. In either case, the OPEN cannot proceed past this point.

After the decision above, the MCP next searches the directory on system disk or on a user pack for a file identified by the names in FPB.MFID and FPB.ID. If the file is not found, the action is identical to that described above. If the file is found, further decisions are necessary.

If the LOCK bit is set in the OPEN adverb, there may be no other users who are writing to the file. There may be other users of the file, but none of them may be using the file for output. If the LOCKOUT bit is set in the OPEN adverb, there may be no other users of the file; the user who is presently opening the file must be the sole user. If these conditions are not met, an operator message is displayed and, depending upon the setting of the REPORT FILE LOCKED bit in the adverb, the user is either suspending or notified of the condition.

Assuming that all of the conditions specified are met satisfactorily, the File Header is brought into memory by the MCP, if it is not there already, and the user count field in the header is incremented. If the OUTPUT bit in the adverb is set, the output user count field is also incremented.

At this point in the OPEN processing, all of the paths converge. If the file is not a disk file, a device has been assigned to it. If the file is a disk file, the File Header is in memory and its associated disk areas, if any, are essentially "assigned" to the file. The File Information Block (FIB) must now be constructed.

Construction of the FIB is a rather mechanical process. After initializing certain fields in the I/O Assignment Table (IOAT), memory to contain the FIB is allocated, if this has not already

occurred. As mentioned in a prior section, the amount of memory allocated for an FIB is dependent upon the type of device assigned.

The FIB itself is constructed from information contained in the FPB, from information in the Disk File Header, and from the parameters passed in the OPEN communicate. After this occurs, the I/O descriptors are constructed. Memory space which contains the I/O descriptors and their associated buffers is allocated with the FIB, such that the FIB contains not only the file information but also the descriptors and buffer areas. The memory necessary is then a contiguous block. This statement does not apply to Data Management System buffers, which are allocated separately.

For serial, input only files, each I/O descriptor is initiated as it is constructed. The buffers are hence "pre-filled" by the operating system when the file is opened. This is true of all files except those assigned to a reader-sorter and those assigned to a data recorder where the user has specified that stacker selection is to be performed on the cards.

For output files which are not assigned to disk, labels are constructed and written according to the user's specifications. Tape labels are discussed in the portion of this document which describes Magnetic Tape Management. For input files, the device assigned will be positioned such that the first READ issued by the program yeilds the first physical record from the device. This is often accomplished by the Open routine.

If the user has requested that translation be performed by the software, memory to contain the Translation Table specified by the user is allocated by the Open routine. The Translation Table is also brought into memory by the Open routine and pointers to it are constructed in the FIB. If the specified Translation Table is not present on disk at the time of the Open, the program is suspended and an appropriate operator message is displayed.

If the LOG System Option is set, entries are made in the log when the file is opened. The FPB for the file is also the log entry and certain fields therin are updated and modified. At the conclusion of the Open processing, control is returned to the user if OPEN was invoked by a communicate. In all languages except COBOL, OPEN may be invoked by a READ, WRITE or SEEK communicate. If this was the case, control is returned to the appropriate communicate handler via the systems processor queues.

CLOSE

```
CT.VERB      09
CT.OBJECT    FILE.NUMBER
CT.ADVERB    BIT
             0      REEL
             1      RELEASE
             2      PURGE
             3      REMOVE
             4      CRUNCH
             5      NO REWIND
             6      OVERRIDE NAME CONVENTION AND SECURITY
             7      LOCK
             8      IF NOT CLOSED
             9      ROLLOUT
             10     AUDIT SWITCH
             11     TERMINATE
```

The CLOSE communicate allows the user to specify the dispensation of files that he has created. Should a program terminate without performing a Close on any file that he has opened, the MCP will assume that the device assigned to the file is to be returned to the system's resources and that the data contained in the file should not be retained.

A second purpose of the Close routine is to bring the I/O activity on a device, which happens somewhat asynchronously with a program's processing, to an orderly halt. It also returns any memory assigned to the file to the system. Clearly, an I/O descriptor and buffer area cannot be returned to available memory until the I/O operation it describes is complete. In order to accomplish this, it is often necessary for the Close routine to give up control of the processor and regain it when certain I/O operations go to completion.

The first test performed by the Close routine is whether or not the file has ever been opened. A CLOSE communicate issued for such a file is considered a programming error and the program will be discontinued at this point. This is done primarily to inform the object programmer of the fact that there is something is wrong.

The second test performed by the Close routine is whether or not the file is open now. It is considered a programming error if a user requests a Close on a file that is already closed, as opposed to never having been opened, if the IF NOT CLOSED bit is not set in the CLOSE communicate adverb. The program will be automatically discontinued if this error is detected. If the IF NOT CLOSED bit is set in the adverb and the file is already closed, control is returned to the user program through the

normal processor queue mechanism. All other bits in the adverb will have no effect and the file is not closed a second time.

Before proceeding with a description of the mechanics of the CLOSE communicate, it will be beneficial to explain the function of the various bits in the adverb. The REEL bit is used on files which are assigned to magnetic tape only. It is ignored by the code if the file is assigned to any other device. It causes the MCP to close the reel of magnetic tape that is currently being processed. The file will be closed and the reel will be locked by the MCP. If the user program issues another OPEN communicate for the file, the next reel of the file, in numerical sequence, will be sought by the Open routine.

It is not necessary for the user program to issue CLOSE REEL communicates when the physical end of the reel is encountered. This is done automatically by the MCP. Reel-to-reel transition is accomplished without the involvement of the user program.

The RELEASE bit in the adverb means that the resources assigned to the file are to be returned to the system. New disk files which are closed with RELEASE will have their assigned disk areas, if any, returned to the list of available disk. Permanent disk files which are closed with RELEASE will have their user count fields decremented but will remain in the disk directory. Devices other than disk will be marked available for use by other jobs, provided their physical status permits.

The PURGE function is applicable to files assigned to disk or tape only. It is ignored if the file is assigned to other devices. If a CLOSE PURGE is performed on a file which is assigned to a tape unit, the tape reel will be rewound and purged, provided it is write-enabled. If it is not write-enabled, CLOSE PURGE will be equivalent to CLOSE RELEASE. For a permanent disk file which is closed with PURGE, the file is removed from the directory, provided the user who is doing the CLOSE is the sole user of the file, and the disk space assigned to the file will be returned to the available table. A new disk file, often known as a "temporary" file, will not yet be entered in the disk directory, but the disk space assigned to it will be returned to the available table also. In the case of a temporary file, there can be only one user and it is not necessary to check user counts before purging.

The LOCK bit in the adverb is intended to be used on files which are assigned to disk and tape only. It is ignored if the file is assigned to other devices. When a file assigned to tape is closed with LOCK, the tape reel is rewound and the unit's status is marked as "Locked" in the IOAT. The unit will not be

available for use by other jobs until the operator intervenes, either by making the unit not ready and then making it ready or by entering a "Ready" message on the SPO. The intended purpose of this function is to prevent the MCP from assigning the unit to other jobs before the operator has had a chance to remove any tape files which may have been created on the unit.

The LOCK bit, when set on a CLOSE directed to a file which is assigned to disk, causes the file to be entered in the disk directory if it is a temporary file, subject to the restrictions below. If the file is a permanent file which is already in the directory, the LOCK function is equivalent to the RELEASE function.

A file may not have its name entered in the disk directory if there is already a file by that name in the directory. A user who attempts to CLOSE LOCK a file, the name of which is already in the directory causes what is known as a "Duplicate Library" condition. The program will be suspended at this point and an operator message describing the conflict will be displayed. The operator must intervene at this point and cause the existing file to be removed, or instruct the MCP to change the CLOSE LOCK communicate to a CLOSE PURGE or CLOSE RELEASE. Syntax is provided to allow this.

The REMOVE bit in the adverb is intended to be used on disk files only. Its function is to allow temporary disk files to be Closed with LOCK without operator intervention. It operates in a manner similar to CLOSE LOCK except that if a Duplicate Library condition arises, the existing file is removed from the directory and the disk space assigned to it is returned to the available table automatically. The function is performed by the MCP with no operator intervention required. The new disk file is then entered into the directory and control is returned to the user.

The CRUNCH bit in the adverb was originally intended for use by the compilers. This restriction is not enforced, however, and it may be used by any program whose source language includes the construct necessary to set the bit in the Communicate format. Its purpose is to return any disk that was requested but not used by the file to the available disk table. The unused disk must lie beyond the End-of-File pointer for the file and the file may have no more than one disk area assigned to it. When a Close with CRUNCH operation is performed on a file, always in conjunction with the LOCK bit, the number of segments per area in the file header is modified by the Close routine such that it is exactly equal to the amount of disk used. The header is then written to disk, per the LOCK bit, and the unused space is returned to the system.

The NO REWIND bit when set is applicable to magnetic tape files only. It causes the magnetic tape to be positioned immediately beyond the last label record written. The unit remains assigned to the program and is not available for use by anyone else. The user then has the option of opening another file in the forward direction, thus creating or continuing a multi-file tape, or of opening the file just written as input in the reverse direction.

The IF NOT CLOSED bit allows a user to close a file that is already closed. Ordinarily, this is a programming error and will result in the user's being terminated by the MCP, as described in a prior paragraph. This bit is merely a means of avoiding termination.

File Information Blocks, I/O descriptors and buffer areas require substantial amounts of memory. The ROLLOUT bit in a CLOSE Communicate was provided to allow a user to temporarily close a file, leaving the associated device assigned to the program, and have the FIB stored on disk so that it does not waste memory space. When the file is reopened, it is merely a matter of reading the FIB in from disk, updating certain fields therin, and proceeding. This is often quicker than recreating the entire FIB and it elimiates the possibility of another program gaining control of the peripheral device in the interim.

The TERMINATE bit in the CLOSE adverb is set when the MCP's termination routines call the Close routine. This occurs only when a program terminates, normally or abnormally, and the peripheral devices are still assigned to the program.

The MCP insures that the external names associated with a file assigned to disk are proper names when the file is closed. When a compiler closes the code file it has generated, the external names of the file are inserted by the Close routine based upon information supplied by the user in the Compile Control Card. Also, the MCP will not allow a disk file to be closed with LOCK with a blank multi-file ID. The internal name of the file will be inserted in FPB.MFID and an operator message will be printed when this is attempted.

The FPB.LOCK boolean, if set, will cause the LOCK bit in the CLOSE adverb to be turned on when the file is closed, provided the TERMINATE bit is also set. This causes the file to be entered in the disk directory and was added as an aid to debugging. Occasionally, when a program has a fatal error, disk files that the program was using at the time are helpful to the object programmer in determining what caused the error. Locking the file in the directory enables the programmer to look at the data he was processing when the error occurred.

The CRUNCH bit in the adverb will be turned on automatically by the MCP if the file being closed is a backup file, a pseudo-deck or a code file. The LOCK bit will be turned on if the file is a backup file or a code file and if it should be entered in the directory. This latter case can only be determined from information contained on the Compile card which is not readily available to the compiler.

Any file which was opened with the output bit set in the OPEN adverb, any file to which the user may have been issuing WRITE communicates, requires some special attention by the MCP during the Close procedure. Since physical I/O operations happen asynchronously with the user program, output I/O operations may have been initiated or marked ready for initiation and be incomplete or not even in process at the time the MCP receives the CLOSE communicate. The actual Close operation must therefore wait for the completion of all output I/O operations associated with the file that is being closed.

Input files present some similar problems. Since all of the user's buffers are filled when the file is opened, provided the file is accessed serially, and since the MCP attempts to stay ahead of the user program in initiating I/O operations, as soon as the user has read all of the records from a buffer, physical I/O operations may be in process or marked ready for initiation when the file is closed. In the case of an input file, it is not necessary for the MCP to wait for I/O completion. Any operations which have not been physically initiated may be cancelled by removing them from the channel chain. The MCP must wait for the completion of any I/O operations that are already physically in process, but this is a relatively short time period.

In the case of Sequential I/O and Delayed Random disk files, the data in the buffer may have been altered by a Write operation from the user but the I/O descriptor may not yet have been marked ready for initiation. The Close routine will insure that all such buffers are actually written to disk prior to the completion of the Close operation. Similarly, for serial, blocked output files, the user may have done several write operations but not yet filled an entire buffer. The I/O descriptor in this case also will not yet be marked ready for initiation but the buffer will contain data which must be written to the physical media. The Close routine will initiate all such operations and insure that they are completed satisfactorily before allowing the file to be closed.

In order for the events described in the preceding paragraph to occur, the physical media must remain accessible. In other

words. If the unit goes not ready and there are I/O operations which must be completed before a Close can occur, the program will remain in a waiting condition until the unit goes back to a ready condition and the necessary operations are complete. Keyboard syntax is provided, however, to allow the operator to override this restriction. The syntax should be used only when the user program is being aborted. The output data in the buffers will be lost if the syntax is invoked and the data in the file will be suspect. Further, if the device is a magnetic tape, the MCP will not be able to write closing tape marks and labels on the media and an I/O error will result when the tape is read. Possibly, no I/O error will result, which may be worse.

The Close routine next begins operations which are dependent upon the type of device assigned. In the case of a card reader which is closed by a user, the device may contain cards which have not yet been read by the program. The MCP will cause the cards to be passed through the reader, stopping when the device goes not ready or when the next control card is encountered.

In the case of a reader-sorter, code segments would have been marked non-overlayable in memory when the file was opened. These code segments will be marked overlayable by the Close routine, provided the user who issued the CLOSE is the sole user of a sorter. Reader-sorter files may only be closed when the flow of documents is stopped. Also, they may only be Closed with Release. The MCP will interpret all Close operations on sorter files to be Close with Release, regardless of the setting of the RELEASE bit in the adverb.

By far, the most complicated processing occurs when the file is assigned to disk. If the file is a multipack file, the Base Pack must be on-line at the time of the Close. The Close procedure will not proceed past this point if it is not.

The MCP next attempts to do the LOCK function described previously. New disk files will be entered in the disk directory, provided there is no file with an identical name already in the directory.

Existing files in the disk directory cannot be removed under any circumstances if they are in use. Similarly, files classified as "System" files cannot be removed, even though their user-count field in the disk header is zero. The MCP code file being used is an example of such a file. Existing files will be removed by the Close routine if the REMOVE bit is set in the CLOSE adverb or the RMOV system option is set, if the Close routine encounters a duplicate file in its processing. If neither of the above conditions are true, the program will be suspended at this point

and the operator must intervene to resolve the conflict.

If the file is a permanent file and if the user has added records to the file while it was open, the end-of-file pointer in the file header will be adjusted by the Close routine.

If the PURGE bit is set in the adverb and the file is a permanent file or if the file was temporary and is being Closed with Release, the disk space used by the file is returned to the available table.

If the file being closed is assigned to tape, the Close routine writes tape marks and labels on the tape. Also, the Close routine sends a rewind descriptor to the unit, if not prohibited by the type of Close being performed.

The information in the IOAT is updated by the Close routine. Test and Wait for Ready I/O descriptors are re-initiated, if appropriate. All of the user's I/O descriptors are removed from the I/O chain.

Information in the FPB is updated and stored on disk in the working copy of the FPB. Finally, the memory assigned to the file is returned to the system's available memory. Control is returned to the user through the normal processor queue mechanisms.

POSITION (MICRO MCP (BACKUP FILES ONLY))

```
CT.VERB          10
CT.OBJECT        FILE.NUMBER
CT.ADVERB        BIT
                 0      REPORT & RETURN TO USER ON EOF
                 1      REPORT & RETURN TO USER ON PARITY
                 2      REPORT & RETURN TO USER ON INCOMPLETE I/O
                 3-7
                 8      POSITION TO END OF FILE
                 9      CT.1 CONTAINS PRINTER CHANNEL NUMBER
                 10     CT.1 CONTAINS RECORD COUNT AS A FIXED NUMBER
                 11     CT.1 CONTAINS RECORD NUMBER DESIRED
CT.1             DEFINED BY BITS IN CT.ADVERB
REINSTATE.MSG.PTR VALUES
                 0    GOOD POSITION
                 1     END OF FILE (OR END OF PAGE ON PRINTER)
                 2    I/O ERROR
                 3    INCOMPLETE I/O
```

The POSITION communicate allows the user to change the physical and logical position on a file. It is used with serial files only, of course. The file may be assigned to disk, tape or to a printer. The communicate is ignored if the file is assigned to any other device.

Positioning a printer file will be discussed first. If the POSITION communicate is directed to a printer file, CT.1 will contain either a channel number which will correspond to a punch in a carriage control tape, or a number which will specify the number of lines the printer should be spaced. If bit 10 in CT.ADVERB is on, CT.1 will be assumed to contain the channel number. If the bit is off, CT.1 will be assumed to contain the number of lines.

The Position routine will always space the printer the number of lines requested. Due to the design of the B1000 Printer Controls, it spaces the printer two lines per descriptor and, if the number of lines requested was an odd number, issues a space operator for one line to complete the operation. If a channel twelve punch in the carriage control tape is reported anytime during the spacing, End-of-page is reported to the program when the operation is complete. It is therefore possible, though highly inefficient, to cause several pages of paper to be passed through the printer with one POSITION communicate. End-of-page is always reported to the user, if it was reported to the MCP, regardless of whether of not he has included code to handle the situation. Programs are not automatically discontinued if there is no such code.

If CT.1 contains a channel number, and if the channel number is less than twelve, the routine constructs and sends an I/O descriptor to cause the printer to space to the requested channel. If the channel number if CT.1 is twelve or greater, a message is printed on the SPO and the communicate is ignored.

With some restrictions, disk files may be positioned forward or backward a specific number of records or they may be positioned to a specific record number within the file or they may be positioned to the end of the file. The file may be opened for input or for output but it may not be opened for both.

Random disk files and files with variable length records may not be positioned at all. Attempting to do so will result in the MCP automatically discontinuing the program.

Input disk files may be positioned to the end of the file, but may not be positioned beyond. Attempting to do so will result in

the file being positioned to the end of the file. Output disk files may be positioned beyond the end-of-file pointer but may not be positioned beyond the declared physical bounds of the file. The "declared physical bounds" of a file are the number of records per area declared times the number of areas in the file declaration.

Files may not be positioned to a negative record number. Attempting to do so will result in the file being positioned to the first record in the file automatically.

In all cases mentioned above, the first bit in the adverb must be set. Attempting to position a disk file to or beyond the end-of-file pointer or to the first record in the file or a prior record will result in the MCP automatically discontinuing the program if the Report and Return EOF bit is not set. This is applicable regardless of whether the file is opened for input or output purposes.

Files assigned to tape may also be positioned forward and backward, provided the file does not contain variable-length records. Attempting to position such a file will result in the program being discontinued by the MCP. Also, tape files will be positioned to the first record in a file or to the end-of-the-file, provided the first bit in the adverb is set, in the same manner as disk files.

In order to function properly, the MCP maintains a record count for all tape files on a "per reel" basis. When the Position routine receives the communicate, it first computes the record number desired by the program. If the record count desired exceeds the current record count, the tape is positioned in the forward direction to the desired record.

The record count for any reel of tape is set to zero when the file is opened. This is applicable regardless of the type of Close previously performed on the file, if there was a prior Close. Hence, when a tape reel is opened in the reverse direction, Record One is actually the last physical record on the tape. The "forward" direction is therefore defined to be the direction that the tape is currently being passed. When a file is opened reverse, a "Backspace" operation will cause it to move toward the physical end of the reel.

Tape files may not be positioned to the end of the file if the file is opened output or if it is opened reverse or if it is not an ANSII-labeled tape. When a Position to end-of-file occurs, the record count field maintained by the MCP will be lost, since

the I/O operator addressed to the unit will be a "Space to Tape Mark". The record count field must therefore be recovered from the ending label and ANSII labels are the only labels which guarantee that a record count field is present.

The B1000 tape subsystem is capable of spacing tape one physical record per I/O operation or to a tape mark. It is not capable of spacing for a specified number of physical blocks with one I/O descriptor. Hence, spacing to a specific record occurs one block at a time. Irrecoverable I/O errors encountered on any of the blocks will result in the program's being automatically discontinued by the MCP, if the second bit in the adverb is not set. If the second bit is set, the I/O error will be reported to the program and the position communicate will be terminated. At this point, the record count field maintained by the MCP will not be reliable.

If a tape mark is encountered while the MCP is spacing the tape to a specific record, End-of-File will be reported to the program if the first bit in the adverb is set. The program will be automatically discontinued if it is not.

ACCESS FILE PARAMETER BLOCK (FPB)

```
CT.VERB      11
CT.OBJECT    FILE.NUMBER
CT.ADVERB    BIT
             0-10  -
             11    0=READ
                   1=WRITE
CT.1         RECEIVING FIELD BIT LENGTH
CT.2         RECEIVING FIELD BASE RELATIVE BIT ADDRESS
```

The ACCESS.FPB Communicate allows the user access to any of his File Parameter Blocks. The working copy only of the FPB may be accessed by this communicate. The FPB is read directly from disk into the user's run structure. The address and size passed in the communicate must lie wholly within the run structure. The program will be automatically discontinued by the MCP if this rule is violated.

Information is not formatted by the MCP. If the FPB definition in the MCP is changed for any release of the software, it is the user's responsibility to make corresponding changes in his program.

The communicate operation is ignored if CT.OBJECT specifies a file which is non-existent or if the user attempts to read less

than 56 bits of an FPB.

Changes made to the FPB while the file is open will not be effective until the file is opened again. Due to the fact that the Close procedures use fields in the FPB, changing a File Parameter Block while a file is open may result in unpredictable errors and even system halts.

ACCESS FILE INFORMATION BLOCK (FIB)

```
CT.VERB      12
CT.OBJECT    FILE.NUMBER
CT.ADVERB    BIT
             0-10  -
             11    FORMAT
                   0=CHARACTER
                   1=BINARY
CT.1         RECEIVING FIELD BIT LENGTH
CT.2         RECEIVING FIELD BASE RELATIVE BIT ADDRESS
```

The ACCESS.FIB communicate does not really access the entire FIB. It returns only the End-of-File pointer and the type of hardware device assigned to the file. It returns these items in either binary or decimal format. The End-of-File pointer is twenty-four bits or eight bytes. The hardware type is six bits or two bytes respectively.

Programs will be automatically discontinued if the receiving file is not wholly contained within the program's run structure. An ACCESS.FIB communicate for a file that is not open will be ignored.

DATA OVERLAY

```
CT.VERB      13
CT.OBJECT    BASE RELATIVE BIT ADDRESS OF 76 BIT FIELD IN FORMAT OF :
             4 BITS   -
             24 BITS  BEGINNING ADDRESS
             24 BITS  ENDING ADDRESS
             24 BITS  RELATIVE DISK ADDRESS
```

This communicate is issued by programs which are written in SDL and which include paged arrays only. The SDL Compiler generates code which manages the paged array space and this communicate is the means whereby it transfers information in the paged arrays to and from disk.

The area described by the fields listed above must lie wholly within the program's run structure. Violation of this rule will result in the automatic discontinuation of the program.

The relative disk address passed must lie within the disk overlay area allocated to the program. This has been discussed previously under program BOJ facilities.

Due to hardware limitations, the overlay area can be no smaller than 56 bits.

This communicate is not used by COBOL, RPG or any other program written in a source language other than SDL.

This communicate uses the program's overlay descriptor in the Run Structure Nucleus. The program is placed in the WAIT.Q until the I/O operation initiated by the procedure goes to completion. At that time, control is returned to the program through the normal processor queues.

ACCESS DISK FILE HEADER (DFH)

```
CT.VERB          14
CT.OBJECT        BASE RELATIVE ADDRESS OF 30 CHARACTER FILE IDENTIFIER :
                 PACK.ID CAT MFID CAT FID
CT.ADVERB        BIT
                 0-5     -
                 6       OVERRIDE USERCODE NAMING CONVENTION AND SECURITY
                 7       REPORT SECURITY VIOLATION
                 8-9     -
                 10-11   0=WRITE
                         1=READ
                         2=READ & FORMAT IN BINARY
                         3=READ & FORMAT IN CHARACTERS
CT.1             RECEIVING FIELD BIT LENGTH
CT.2             RECEIVING FIELD BASE RELATIVE BIT ADDRESS
REINSTATE.NSG.PTR VALUES
                 0     COMMUNICATE COMPLETE
                 1     FILE NOT PRESENT OR SECURITY VIOLATION AND
                          CT.ADVERB BIT 7=0
                 2     SECURITY VIOLATION AND CT.ADVERB BIT 7=1
```

This communicate allows the user access to disk file headers contained in the system's disk directory. The receiving or sending file described by CT.1 and CT.2 must lie within the program's run structure. If it does not, the program will be automatically discontinued by the MCP. The field in the run structure which specifies the file identifier must be exactly

thirty characters in length and must conform to the fixed format described in the Communicate layout above.

This communicate has four variations as defined by CT.ADVERB. If CT.ADVERB contains a zero or a one, the sending or receiving field is assumed to correspond exactly to the current definition of a disk file header. The "current" definition means the definition used in the actual MCP that is handling the communicate operator, and not the definition used in any subsequent MCP.

If CT.ADVERB is set to zero, certain fields are moved from the program's run structure to the actual disk file header and written to the disk directory. Only selected fields may be written; those not selected are ignored.

If CT.ADVERB is a one, information is moved directly from the file header to the receiving field specified. The move is left-justified with zero fill. The entire file header may be read in this manner.

If CT.ADVERB is a two or a three, the fields listed in the table below only are moved to the program's run structure. The formatted move also occurs left-justified with no filling. If the receiving field is not sufficiently long, the move is merely truncated from the right.

FIELD NAME	LENGTH (BITS)	LENGTH (CHARACTERS)
OPEN.TYPE	24	1
NO.USERS (Number of Users)	24	2
RECORD.SIZE	24	4
RECORDS.PER.BLOCK	24	4
EOF.POINTER	24	8
SEGMENTS.PER.AREA	24	8
USERS.OPEN.OUTPUT	24	1
FILE.TYPE	24	2
PERMANENT	24	1
BLOCKS.PER.AREA	24	6
AREAS.RQST (Requested)	24	3
AREA.COUNTER	24	3
SAVE.FACTOR	24	3
CREATION.DATE	24	5
ACCESS.DATE (Last)	24	5
REC.SIZE	-	5
MPF (Multi-Pack File)	1	1
PROTECTION	2	1
PROTECTION.IO	2	1

If the file is not present in the disk directory, the program is notified by inserting a one in the RS.REINSTATE.MSG.PTR. In either case, control is returned to the program through the normal processor queue mechanism.

FIND/MODIFY (DM)

```
CT.VERB      15
CT.OBJECT    INVOKE NUMBER & PATH NUMBER OF THE PATH-NAME
CT.ADVERB    BIT
             0      RETURN LIST HEADS (REORG ONLY)
             1      RETURN LOGICAL ADDRESS (REORG ONLY)
             2      DM.STATUS FORMAT
                    0=BINARY
                    1=4-BIT DECIMAL
             3      ON EXCEPTION
             4      -
             5      MODIFY
             6-10   SELECTION EXPRESSION
                    0  NEXT
                    1  PRIOR
                    2  FIRST
                    3  LAST
                    4  NEXT AT
                    5  CURRENT
                    6  AT
             11     DATA SET SELECTION EXPRESSION
CT.1         DM.STATUS REGISTER BIT LENGTH
CT.2         DM.STATUS REGISTER BASE RELATIVE BIT ADDRESS
CT.3         DATASET RECORD WORK AREA BIT LENGTH
CT.4         DATASET RECORD WORK AREA BASE RELATIVE BIT ADDRESS
CT.5         SEARCH KEY (CAT OF COMPONENT NAMES) BASE RELATIVE BIT ADR.
CT.6         INVOKE NUMBER & PATH NUMBER OF DATASET-NAME
```

Refer to P.S. 2212 5470.

STORE (DM)

```
CT.VERB      16
CT.OBJECT    INVOKE NUMBER & PATH NUMBER
                    (SUBSET IF INSERT)
CT.ADVERB    BIT
             0      INSERT
             1      -
             2      DM.STATUS FORMAT
                    0=BINARY
                    1=4-BIT DECIMAL
             3      ON EXCEPTION
             4      BEGIN TRANSACTION (NOT INSERT)
             5      INCLUDES.LIST.HEADS (REORG ONLY)
             6      END TRANSACTION (NOT INSERT)
```

```
                7       NO AUDIT (BEGIN OR END TRANSACTION ONLY)
                8       SYNC (END TRANSACTION ONLY)
                9       -
                10      STORE INDEXES ONLY (REORG ONLY)
                11      PSEUDO CREATE (REORG ONLY)
  CT.1          DM.STATUS REGISTER BIT LENGTH
  CT.2          DM.STATUS REGISTER BASE RELATIVE BIT ADDRESS
  CT.3          DATASET RECORD WORK AREA BIT LENGTH (NOT INSERT)
                INVOKE NUMBER & PATH NUMBER OF DATASET (INSERT)
  CT.4          DATASET RECORD WORK AREA BASE RELATIVE BIT ADDRESS
                (NOT INSERT)
```

Refer to P.S. 2212 5470.

DELETE (DM)

```
  CT.VERB       17
  CT.OBJECT     INVOKE NUMBER & PATH NUMBER
                (SUBSET IF REMOVE)
  CT.ADVERB     BIT
                0       REMOVE
                1       -
                2       DM.STATUS FORMAT
                        0=BINARY
                        1=4-BIT DECIMAL
                3       ON EXCEPTION
                4-11    -
  CT.1          DM.STATUS REGISTER BIT LENGTH
  CT.2          DM.STATUS REGISTER BASE RELATIVE BIT ADDRESS
  CT.3          DATASET RECORD WORK AREA BIT LENGTH (NOT REMOVE)
                INVOKE NUMBER & PATH NUMBER OF DATASET (REMOVE)
  CT.4          DATASET RECORD WORK AREA BASE RELATIVE BIT ADDRESS
                                                    (NOT INSERT)
```

Refer to P.S. 2212 5470.

CREATE/RECREATE (DM)

```
  CT.VERB       18
  CT.OBJECT     INVOKE NUMBER & PATH NUMBER
  CT.ADVERB     BIT
                0       -
                1       RECREATE
                2       DM.STATUS FORMAT
                        0=BINARY
                        1=4-BIT DECIMAL
                3       ON EXCEPTION
                4-11    -
  CT.1          DM.STATUS REGISTER BIT LENGTH
  CT.2          DM.STATUS REGISTER BASE RELATIVE BIT ADDRESS
```

```
CT.3        DATASET RECORD WORK AREA BIT LENGTH
CT.4        DATASET RECORD WORK AREA BASE RELATIVE BIT ADDRESS
```

Refer to P.S. 2212 5470.

SWITCH.TAPE.DIRECTION

```
CT.VERB     19
CT.OBJECT   FILE.NUMBER
CT.ADVERB   BIT
            0-7   NOT USED
            8-11  0 = READ FORWARD
                  1 = READ REVERSE
                  4 = WRITE
REINSTATE.MSG.PTR VALUES
            0     GOOD SWITCH
            1     FILE NOT OPEN
            2     WRONG DIRECTION OR NOT A TAPE FILE
            3     END OF FILE
```

This operator was added to facilitate the implementation of the Tape Sort feature. It has found use in other applications since that time. Essentially, it merely changes the direction of a tape file without time-consuming Close and Open invocation.

There is no way that this communicate can cause discontinuation of a program. All errors are merely reported to the program. The file may be changed from input to output, provided it is not being read in the reverse direction. Direction may be changed on the same communicate which changes the I/O mode. In other words, a file may be changed from input and reverse to output and forward with one communicate.

Buffers are filled by the MCP as a function of this communicate. No fields in the FIB are changed, however. Consequently, use of this communicate is not practical on blocked files.

This communicate will not function if one of the file's buffers has already encountered the physical end of the file.

TERMINATE (STOP RUN)

```
CT.VERB     20
```

This Communicate calls the Terminate Procedure directly. The Terminate Procedure is also called when a program is being

discontinued. There is very little difference between a normal terminate procedure, where the routine is called via a Communicate and an abnormal one, where the procedure is called by the MCP to discontinue a program.

Programs may not be terminated if they are using a reader-sorter and the device is operating. In this case, the terminate rountine will wait until the flow is stopped on the sorter and then proceed with the termination. This is due to the fact that High-Priority Interrupts from the sorter can only be handled in code written exclusively for that purpose. At any rate, all of this will be transparent to the user and the program will be terminated, though not necessarily at the time the Terminate Procedure is first invoked.

A similar situation exists if the program has Data Management operations in process. I/O Complete on such an operation can only be handled by Data Management code, and the Terminate Procedure will be forced to wait for completion of any such operations.

The Terminate Procedure may also have to wait for Roll-in and Roll-Out operations to be completed. The program must be present in memory before it can be terminated.

As mentioned previously, all of the conditions listed to this point are transparent to the user. The Terminate Procedure has its own mechanisms for waiting for such events to be complete. No action is required on the part of the user.

The queue of keyboard messages entered via the Accept response will be purged of any messages intended for this program at this point. Refer to the Software Operational Guide explanation of the "AX" message for details on this queue.

At this point, the Terminate Procedure will wait for any code or data overlays which may be in process to go to completion. The Terminate Procedure, if it must wait for such an event, yields control to the outer loop of the MCP. The Procedure will be continued when the I/O goes to completion.

Any I/O operation which was initiated by the program and which did not use the normal File I/O mechanism will be halted and delinked from the channel chain at this point. Examples of such operations are disk I/O initiated by the Disk Initialization utilities and all Data Communications I/O operations. Temporary disk storage obtained by the MCP to execute the program is

returned to the disk available table.

The Terminate Procedure next proceeds to close all the files which are associated with the program and which are not yet closed. A file which is closed by a user with no bits set in the Close Adverb or with the NO REWIND bit set in the adverb is not considered closed by the MCP. The unit, in these cases, remains assigned to the program, even though there can be no I/O in process for the file. The unit must be returned to the list of available resources when the program terminates. Files are always Closed with Release by the Terminate Procedure, except when the file is assigned to disk and FPB.LOCK is set.

The Terminate Procedure next performs those functions associated with memory assigned to the program. The code segment dictionary user count is decremented. If it becomes zero, memory occurpies by the code segments and the segment dictionary is returned to the available memory list. Similarly, the user count for the Interpreter used by the program is decremented. If it becomes zero, memory occupied by the interpreter and its segments is also returned. If the interpreter was partially or totally resident in M-Memory, it is removed. This may result in a change in the mode of M-Memory management. If so, it is performed at this point.

Similar functions are performed on any Intrinsic Code the program may have been using. The user count for the Intrinsic File and for the Code file itself are decremented and stored in the disk file header in the disk directory.

If the Log option is set, the Log is updated at this point. In addition to the type of termination, a count of code overlays, a count of data overlays, the current time and date and the amount of processor time used by the program are stored in the Log.

The program's overlay descriptor is removed from the disk chain, a SPO message is printed if the EOJ option is set, and if this program was executed by another using the PROGRAM.CALL communicate, the calling program is marked ready to run. Memory occupied by the program's run structure is returned to the available pool. The number of jobs running is decremented. A bit is set which will cause the next execution of the OUTER.LOOP to check the active job schedule.

If any programs are in the Waiting schedule and are waiting for the successful termination of this program, they are moved to the Active schedule, provided this is a normal termination. If this program was a "Compile and Go" or a "Compile and Save", the code

file generated is placed in the active schedule.

FREE (DM)

```
CT.VERB      21
CT.OBJECT    INVOKE NUMBER & PATH NUMBER
CT.ADVERB    BIT
             0-1   -
             2     DM.STATUS FORMAT
                   0=BINARY
                   1=4-BIT DECIMAL
             3     ON EXCEPTION
             4-11  -

CT.1         DM.STATUS REGISTER BIT LENGTH
CT.2         DM.STATUS REGISTER BASE RELATIVE BIT ADDRESS
```

Refer to P.S. 2212 5470.

TIME/DATE/DAY

```
CT.VERB      22
CT.OBJECT    BASE RELATIVE BIT ADDRESS OF WHERE TO PUT THE RESULT
CT.ADVERB    BIT
             0     1=DATE REQUESTED
             1-2   FORMAT
                   0 YY/DDD   (JULIAN)
                   1 MM/DD/YY
                   2 YY/MM/DD
                   3 DD/MM/YY
             3-4   REPRESENTATION
                   0 BINARY
                   1 4-BIT DECIMAL
                   2 8-BIT DECIMAL
             5     1=TIME REQUESTED
             6-7   FORMAT
                   0 COUNTER
                   1 HH:MM:SS.S (24-HOUR CLOCK)
                   2 HH:MM:SS.S TT (12-HOUR CLOCK, TT=AM/PM)
             8-9   REPRESENTATION
                   0 BINARY
                   1 4-BIT DECIMAL
                   2 8-BIT DECIMAL
             10    1=TODAYS.NAME REQUESTED
             11    -
NOTE : TODAYS.NAME RETURNS 9 CHARACTERS LEFT JUSTIFIED
```

FORMAT	BINARY	4-BIT DECIMAL	8-BIT DECIMAL
YY/DDD (JULIAN)	7+9=16	8+12=20	16+24=40
MM/DD/YY	4+5+7=16	8+8+8=24	16+16+16=48

YY/MM/DD	7+4+5=16	8+8+8=24	16+16+16=48
DD/MM/YY	5+4+7=16	8+8+=24	16+16+16=48
COUNTER	20	24	48
HH:MN:SS.S	5+6+6+4=21	8+8+8+4=28	16+16+16+8=56
HH:MM:SS.S TT	4+6+6+4+16=36	8+8+8+4+16=44	16+16+16+8+16=72
TODAYS.NAME			72 (9 CHAR, LEFT JUST.)

INITIALIZER I/O

```
CT.VERB      23
CT.OBJECT    BASE RELATIVE ADDRESS OF
                   6 BYTE UNIT MNEMONIC
                   OR
                   I/O DESCRIPTOR
CT.ADVERB    VALUE
             0     ASSIGN UNIT TO THIS PROGRAM
             1     RELEASE UNIT
             2     INVALID
             3     LINK IN THE I/O DESCRIPTOR AND INITIATE
             4     INVALID
REINSTATE.MSG.PTR VALUES
             IF CT.ADVERB=0 THEN
             PORT, CHANNEL AND UNIT OF DEVICE REQUESTED
                  PORT BIT (3)
                  CHANNEL BIT (4)
                  FILLER BIT (1)
                  UNIT BIT (4)
             ALL OTHER CASES
                  0   GOOD COMMUNICATE
                  1   DISPATCH TO INVALID PORT OR CHANNEL
```

This communicate is intended for in-plant use only. Anyone outside of Santa Barbara Plant who attempts to use this communicate does so at his own risk. The communicate format and function may be changed from time to time. No notice of such change will be supplied to any user prior to the change. Such information will be available on request.

Released MCP's will not allow a Write descriptor to be initiated on system disk. Attempting to do so will result in the program's being DS-ed.

The MCP does not assure that the A and B addresses in the I/O Descriptor are bounded by the program's Base/Limit registers. It is the programmer's responsibility to do this. Failure to do so will result in unidentifiable system halts.

To use this communicate, the programmer should first issue it with CT.ADVERB set to zero and CT.OBJECT pointing to a

six-character unit mnemonic. If the requested unit is not available for any reason, the calling program will be DS-ed. If the unit is available, it will be assigned to the calling program. It is possible to read any unit without requesting that the unit be assigned to you.

After the unit is assigned to the program, the communicate may be issued with CT.ADVERB set to two or three. The MCP copies the I/O Descriptor outside the base-limit before it links into the chain. When the I/O completes, the IO.ACTUAL.END and IO.RESULT are moved back into the base-limit area. When the I/O operation is completed, the I/O descriptor is removed from the associated channel chain. In order to again execute the I/O, the program must issue another communicate with CT.ADVERB set to two or three.

The program should issue the communicate with CT.ADVERB set to one before it goes to end-of-job.

It should be emphasized that this communicate was added to the MCP for purposes of on-line pack initialization only and is intended for use only by that program, in the form supplied by Santa Barbara Plant. Requests for maintenance or support from any other source will be ignored.

MAIL (SNOOZE)

CT.VERB	24
CT.OBJECT	LENGTH OF TIME IN 10THS OF A SECOND
FUNCTION	PROGRAM IS PUT TO SLEEP FOR SPECIFIED LENGTH OF TIME

ZIP

CT.VERB	25
CT.OBJECT	-
CT.ADVERB	-
CT.1	MESSAGE AREA BIT LENGTH
CT.2	MESSAGE AREA BASE RELATIVE BIT ADDRESS
REINSTATE.MSG.PTR VALUES	
0	NO ERRORS IN ZIP TEXT
1	ZIPPED INVALID CONTROL CARD

This communicate provides a means for programs to pass control cards and keyboard messages to the MCP. There are no restrictions on the messages which may be passed. No messages are returned to the program by this procedure; the only information returned is an indication of whether or not the syntax of the control instruction was valid.

Any program placed in the schedule as a result of a control message received from a ZIP Communicate will execute asynchronously with and independently of the program which executed the ZIP, unless programmatic means for synchronizing the two programs are provided in the programs.

If the control instructions passed via the ZIP communicate are invalid, a message is printed on the SPO to inform the system operator of the occurrence. This is necessary, since invalid control instructions can result in incorrect operational behavior.

ACCEPT

```
CT.VERB       26
CT.OBJECT     -
CT.ADVERB     BIT
              0      RETURN IF NO MESSAGE
              1-11   -
CT.1          MESSAGE AREA BIT LENGTH
CT.2          MESSAGE AREA BASE RELATIVE BIT ADDRESS
REINSTATE.MSG.PTR VALUES
              0      MESSAGE OF LENGTH ZERO
         @FFFFFF@    NO MESSAGE PRESENT
         ANY OTHER VALUE LENGTH OF MESSAGE IN BITS
```

This communicate was provided as a means of implementing the COBOL ACCEPT verb. Its use is not restricted to programs written in a particular language; it may be used by any program.

The receiving field must lie within the bounds of the program's run structure. The program will be forced to wait for an operator response if bit one in the adverb is a zero and if there is no message in the Accept Queue for the program. Control is returned to the program through the normal processor queues when a response from the operator is received.

Messages are moved to the receiving field left-justified with blank fill.

DISPLAY

```
CT.VERB       27
CT.OBJECT     -
CT.ADVERB     BIT
              0-10   -
```

```
                11     0=CRUNCH BLANKS OUT OF MESSAGE
                       1=PRINT MESSAGE AS IS
  CT.1          MESSAGE AREA BIT LENGTH
  CT.2          MESSAGE AREA BASE RELATIVE BIT ADDRESS
```

This communicate was provided as a means of implementing the COBOL DISPLAY verb. Like ACCEPT, it may be used by any program. It serves merely to print the message described by CT.1 and CT.2 on the SPO.

If bit eleven in the adverb is set, the MCP will reformat the entire message such that non-blank fields in the message are separated by no more than one blank. This is merely a convenience for the user programmer which may be used when the spacing of the words in the message upon the SPO is not important.

USE/RETURN

```
  CT.VERB       28
```

This communicate is not implemented. If received, it is ignored and control is returned to the user.

SORT HANDLER

```
  CT.VERB       29
  CT.OBJECT     BASE RELATIVE ADDRESS OF SORT INFORMATION TABLE
  CT.ADVERB     BIT (12)
                1 -  SORT.RESTART
                2 -  SORT.DUPCHECK
                3 -  SORT.W1.PID
                4 -  SORT.W2.PID
                5-12 FILLER
  CT.1          BASE RELATIVE BIT ADDRESS OF SORT KEY TABLE
  CT.2          INPUT FILE.NUMBER OR ADDR OF MERGE.INPUT.TABLE IF MERGE
  CT.3          OUTPUT FILE.NUMBER
  CT.4          TRANSLATE FILE.NUMBER OR NOT 0
  CT.6          DATA.ADDRESS (DELETE.KEY.TABLE)
  CT.7          IF (SORT.W1.PID := W1.PID.FLAG) THEN
                   DATA.ADDRESS (W1.PID) ELSE 0
  CT.8          IF (SORT.W2.PID := W2.PID.FLAG) THEN
                   DATA.ADDRESS (W2.PID) ELSE 0
```

This communicate provides a means for the user program to call the Sort Intrinsic. For details on the implementation, refer to the proper Sort or Merge Product Specification.

SDL TRACE

```
CT.VERB      30
CT.OBJECT    TRACE FLAGS
```

This communicate is used by all interpreters which include Trace capabilities. Its use is not restricted to SDL only. The communicate is merely a means of turning the Trace on and off. The print line is passed via a type 61 interrupt from the interpreter. The code invoked by this interrupt is little more than a call on the SDL Read/Write Procedure in the MCP.

EMULATOR TAPE (MICRO MCP)

```
CT.VERB      31
CT.OBJECT    FILE.NUMBER
CT.ADVERB    BIT
             0-2     OP.CODE
                        0 = READ
                        1 = WRITE
                        2 = SPACE
                        3 = REWIND
                        4 = TEST
             3-8     OP.CODE.VARIANT
                 3 = REVERSE (READ, SPACE), ERASE (WRITE),
                     TEST.WAIT.READY.NOT.REWIND (TEST)
                 4 = ONE.RECORD (SPACE), TAPE.MARK (WRITE),
                     TEST.WAIT.NOT.READY (TEST)
                 5 = ODD.PARITY (READ, SPACE, WRITE)
                 6 = NOISE (READ, SPACE)
               7-8 = NOT USED
             9-11    SCHEDULING.VARIANTS
                 9 = FETCH.RESULT
                10 = DONT.WAIT
                11 = REPORT AND RETURN ON IO ERROR

CT.1         USER TAPE BUFFER BIT LENGTH
CT.2         USER TAPE BUFFER BASE RELATIVE ADDRESS
CT.3         USER ERROR MASK (BIT SET IMPLIES USER WILL HANDLE THE
                              CORRESPONDING ERROR)
             BIT
             0        (MAY NOT USE)
             1        (MAY NOT USE)
             2        NOT READY
             3        PARITY (NOT ON TEST)
             4        ACCESS (NOT ON TEST)
             5        TRANSMISSION (ON TEXT ONLY)
             6        END.OF.TAPE
             7        BEGINNING.OF.TAPE
             8        WRITE.LOCK.OUT
             9        END.OF.FILE (NOT ON TEST), UNIT.PRESENT (ON
                        TEST)
```

	10	REWINDING
	11	TIME.OUT (NOT ON TEST)
	12-16	(MAY NOT USE)
	17	SHORT.RECORD
	18	LONG.RECORD
	19	DROPOUT
	20	INITIATE.LATE
	21	(MAY NOT USE)
	22	TRANSMISSION.ERROR.MEC
	23	TRANSMISSION.ERROR.MTC

CT.4 BASE RELATIVE ADDRESS OF USER'S 48 BIT RESULT
BIT 0-23 OF RESULT CONTAIN THE RESULT DESCRIPTOR
BIT 24-47 OF RESULT CONTAIN THE ACTUAL LENGTH

REINSTATE.MSG.PTR VALUES

0 = RESULT RETURNED
1 = IO.ERROR
2 = RESULT NOT AVAILABLE

This communicate was added so that the Emulators of second-generation hardware produced by Santa Barbara Plant might be operated under control of the MCP. It was necessary to add a new communicate to do this, since programs written for these machines routinely manipulate magnetic tape in manners which violate the rules of the MCP's logical I/O mechanisms. The normal file mechanisms in the MCP, which were promulgated upon the specifications of the COBOL language, are certainly inadequate to allow all of the many tape operations which were common to second generation machines.

Essentially, the procedure builds an I/O descriptor according to the specifications passed by the communicate format and initiates it. The Emulator program is not allowed to execute until the I/O operation goes to completion. The procedure is re-entered at completion of the operation and the program is then allowed to continue.

The procedure first tests to see if the file is Open. If it is not, the Open procedure is called directly from the communicate and control is returned to it when the open completes. At this point, the procedure continues provided the open was successful. If it was not, the emulator program would have been placed in one of the processor queues and marked waiting. In the latter case, the communicate procedure merely returns control to the outer loop of the MCP.

The procedure next performs minor editing on the files passed in the communicate format. If the operator request involves a data transfer, the buffer area described must lie wholly within the bounds of the user's run structure. Due to hardware

restrictions, certain variants and combinations thereof are invalid for certain operation codes. The variants here are those passed in bits three through seven of the communicate adverb. Validity of the variants is checked by the procedure. If the user has violated either the bounds check or the variant check, the program is automatically discontinued by the MCP.

The procedure next constructs an I/O Descriptor which corresponds to that requested by the user. The I/O Descriptor is outside the user's run structure. A full tape file FIB is allocated, along with space for one I/O descriptor, by the Open Procedure. This is done even though many of the fields in it are not used by the Emulator Tape Handler routines directly. Most of the fields are required by the Close Procedure.

The requested I/O operation is then initiated and the program is marked waiting for its completion. Control is returned to the outer loop of the MCP at this point and the MCP is free to service other users.

When the I/O operation completes, the procedure is again invoked. It first moves the result descriptor received with the I/O concatenated with a twenty-four bit field which will specify the actual length of the operation just completed. The length of the operation is specified in bits. Also, prior to doing the move of the result descriptor, the procedure verifies that the receiving field is within the run structure of the program. If it is not, the program is automatically discontinued.

If the exception bit is set in the result descriptor, the procedure determines if the exception condition is one that the user has included code to handle himself. If it is, control is returned to the user through the normal processor queue mechanism. If the user does not have code to correct the error himself, the procedure calls the MCP's I/O Error procedure directly. I/O Error will then retry a number of times and eventually return control to the Emulator Tape Handler. If the error was irrecoverable, the program is discontinued. Otherwise, control is returned to the user.

COBOL PROGRAM ABNORMAL END

CT.VERB 32

This communicate was added so that job streaming may be terminated at the discretion of the user. It functions exactly as the STOP communicate does except that instead of a standard End-of-Job message, it causes "COBOL ABNORMAL END" to be printed

on the SPO. Also, any programs that are in the Waiting Schedule waiting for this job to finish will not be moved to the active schedule by the abnormal termination.

SORI EOJ

```
CT.VERB     33
CT.OBJECT   FILE.NUMBER
CT.ADVERB   CLOSE TYPE
CT.1        END-OF-FILE POINTER
CT.2        RECORD SIZE
```

This communicate serves to terminate, in a normal manner, the Sort and Merge Intrinsics and to return control to the calling program. For details on its operation, refer to the appropriate Sort or Merge product specification.

FREEZE/THAW RUN STRUCTURE

```
CT.VERB     35
CT.OBJECT   BIT 0 (HIGH ORDER BIT)
            0=THAW
            1=FREEZE
```

Depending upon the functions being performed by a user program, it may not be permissable for the MCP to change the memory location of the program's run structure or to roll it out to disk. The most obvious example of this is the Disk Initialization Utilities, which have actual I/O Descriptors within their run structure. There are several other such cases.

The field in the Run Structure Nucleus, RS.TEMPORARY.FREEZE, gives notice to the MCP's Roll Out procedures that this run structure may not be moved. This communicate provides a programmatic means of bumping and decrementing that field.

COMPILE CARD INFORMATION

```
CT.VERB     36
48 BITS     SDL DESCRIPTOR (WHERE TO PUT INFO) IN FORMAT :
            16 BITS=LENGTH
            24 BITS=ADDRESS
            RETURNS COMPILE CARD INFO IN FOLLOWING FORMAT :
            #CHARS  INFO
            .....   ........................................
            30      OBJECT NAME
            02      EXECUTE TYPE
            10      PACK.NAME OF THE RUNNING PROGRAM
```

```
30        INTERPRETER NAME OF THE RUNNING PROGRAM
10        INTRINSIC NAME (PACK & FAMILY)
02        PRIORITY
06        SESSION
06        JOB NUMBER
20        1ST & 2ND NAMES OF RUNNING PROGRAM
07        CHARGE NUMBER
01        FILLER
36 BITS   DATE COMPILED
04 BITS   FILLER
10        USERCODE
10        PASSWORD
04        PARENT JOB NUMBER
20        PARENT QUEUE IDENTIFIER
01        LOG SPO
04        SECONDS BEFORE DECAY
01        PRIVILEGED
```

This communicate returns selected fields from the working copy of the Program Parameter Block to the user's run structure. The receiving field described by the SDL Descriptor must lie wholly within the run structure. The program will be automatically discontinued if it does not. The fields returned are presented in the fixed format shown in the table above.

DYNAMIC MEMORY BASE

```
CT.VERB     37
            VALUE IS RETURNED IN COMMUNICATE MESSAGE POINTER AS
            SELF RELATIVE DESCRIPTOR
```

This communicate returns the relative address of the dynamic, or overlayable, area in the run structure. It is intended for use by programs written in SDL only.

MEMORY DUMP TO DISK

```
CT.VERB     38
            USED BY ALL LANGUAGES, INCLUDING SDL.
```

This communicate causes the program's run structure and other pertinent information to be dumped to disk and locked in the directory with a unique name. The information may be processed, formatted and printed later by a normal state program written for that purpose.

Programs which are already in the process of terminating may not be dumped. This is academic in this case, however, since a

program which was terminating could not issue the communicate. Programs which are rolled out to disk will be rolled back in and dumped, via an operator's action, provided sufficient memory is available. Again, if the program were rolled out to disk, it could not possibly have issued the communicate.

The amount of disk which will be required to contain the dump file usually exceeds by a considerable margin that required to contain just the Base/Limit area of the program. In addition to the run structure, the dump file will also contain the File Dictionary, the FIB's and buffers, the Data Dictionary and all data segments, the code dictionary and the code segments which are present in memory at the time, and other miscellaneous information. If sufficient disk is not available, a message will be printed on the SPO and a one will be returned to the program in RS.REINSTATE.NSG.PTR. The program will be allowed to continue processing.

The format of a dump file is as follows:

1. A "Pointer" record, which is described below.
2. The program's run structure and Run Structure Nucleus.
3. The Data Dictionary.
4. Every data segment in the dictionary. Segments which are not in memory will be copied to the dump file from their location on disk.
5. The File Dictionary.
6. Each FIB and its associated I/O Descriptors and buffers. This includes all files that are open and those that are closed with no form of release.
7. The working copy of the Program Parameter Block.
8. The working copy of the initial scratchpad settings.
9. The working copies of all File Parameter Blocks.
10. If the program is written in SDL, the LAYOUT.TABLE.

Control is returned to the program through the normal processor queue mechanism. Actually, the program is marked ready to run after the scratchpad is copied.

The format of the "Pointer" record is:

```
DUMPFILE.NUMBER      BIT(24)
TIME.OF.DUMP         BIT(36)  % Julian Date plus Time
MCP.DATE             BIT(16)  % MCP Version Date
RELEASE.MCP          BIT(4)
LIMIT.REGISTER       BIT(24)
DATA.DIC.PTR         BIT(24)  % Relative Disk Address
DATA.SEGS.PTR        BIT(24)  % Relative Disk Address
FIB.DIC.PTR          BIT(24)  % Relative Disk Address
FIB.PTR              BIT(24)  % Relative Disk Address
PPB.PTR              BIT(24)  % Relative Disk Address
NET.CONTROL.MACRO    BIT(4)   % 1 if Data Communications Handler
MCP.RELEASE.LEVEL    BIT(16)
LAYOUT.TABLE.PTR     BIT(24)
LAYOUT.TABLE.SIZE    BIT(24)
DUMP.SYSTEM.ID       BIT(12)
```

GEI SESSION NUMBER

```
CT.VERB      39
CT.OBJECT    SESSION IS PUT INTO RS.REINSTATE.MSG.PTR
```

DC.INITIATE.IO

```
CT.VERB      40
24 BITS      PORT
24 BITS      CHANNEL
24 BITS      BASE RELATIVE ADDRESS OF I/O DESCRIPTOR
```

This communicate provides the capability of initiating I/O descriptors on the data communications equipment which may be attached to a system. The I/O descriptor itself if constructed by the program, which is usually the Data Communications Handler program generated by the NDL Compiler. This is not a requirement, however, and no test for this condition is made by the code in the MCP. The communicate operator may be used by any program whose source language contains the proper syntax.

The program will be automatically discontinued by the MCP if the requested I/O control is not a data communications control or if the control is already in use by another program. Also, if the address of the I/O descriptor does not lie within the program's run structure or if the program attempts to initiate an I/O descriptor with the "high priority interrupt request" bit set, the program will be automatically discontinued.

After the editing described above has been performed, the requested operation is initiated. Control is returned to the user through the normal processor queue mechanism. The program

is not forced to wait for the completion of the operation initiated; control is returned immediately after the initiation.

NDL/MACRO COMMUNICATES

```
CT.VERB      41
CT.OBJECT    INDICATES FUNCTION
DESC1        BIT  1-48   MESSAGE AREA 1
DESC2        BIT 49-96   MESSAGE AREA 2
QUEUE.PTR    BIT 97-106  REMOTE FILE NUMBER OR STATION NUMBER
```

DCWRITE

```
CT.OBJECT    11
DESC1        RESULT AREA
DESC2        DC.WRITE MESSAGE
  NOTE:      NUMBER AT SUBSTR(DESC2,6,2) IS MESSAGE TYPE
             40=FINISH OPEN
             41=NDL/MACRO PRESENT
             42=ATTACH STATIONS TO REMOTE FILE
             43=DETACH STATIONS FROM REMOTE FILE
```

QUICK QUEUE WRITE (REMOTE FILES)

```
CT.OBJECT    12
DESC1        MESSAGE HEADER
DESC2        MESSAGE
RMT.FL       REMOTE FILE TO WHICH THE MESSAGE IS DESTINED
```

QUICK QUEUE WRITE (STATION NUMBER)

```
CT.OBJECT    13
DESC1        MESSAGE HEADER
DESC2        MESSAGE
ST.NR        STATION NUMBER
```

ACCESS USERCODE FILE

```
CT.VERB      42
DESC         BIT 0-47 DESCRIPTOR TO PARAMETER LIST.

  PARAMETER LIST LAYOUT
    MODE     BIT (4)
      0      SET ALL PARAMETERS IN LIST EXCEPT USERCODE AND
               PASSWORD.  THESE MUST BE SUPPLIED TO FIND
               CORRECT ENTRY.
      1      SET ALL PARAMETERS IN LIST EXCEPT INDEX. INDEX
               MUST BE SUPPLIED TO FIND ENTRY.
      2      SET OVERRIDE.  USERCODE AND PASSWORD MUST BE
```

```
                PRESENT TO FIND ENTRY.
          3     SET OVERRIDE.  INDEX MUST BE SUPPLIED TO FIND ENTRY.
          4     ADD ENTRY.  ALL FIELDS HAVE TO BE SUPPLIED.
          5     DELETE ENTRY.  USERCODE AND PASSWORD MUST BE
                SUPPLIED TO FIND ENTRY.
          6     INITIALIZE ALL OVERRIDE BITS.
          7     CHANGE BY USERCODE.  ALL ENTRIES FOR A GIVEN USER-
                CODE CAN BE CHANGED WITH ONE COMMUNICATE.  USER-
                CODE MUST BE PRESENT.  PACK FIELD MUST NOT BE
                EQUAL TO ZERO TO CHANGE IT.  CHARGE NUMBER MUST
                NOT BE EQUAL TO ZERO TO CHANGE IT.  PRIORITY MUST
                NOT BE EQUAL TO ZERO TO CHANGE IT.
          8     DELETE ALL RECORDS FOR A GIVEN USERCODE.  USER-
                CODE MUST BE PRESENT.
          9     SET ALL PARAMETERS IN LIST EXCEPT USERCODE AND
                PASSWORD.  ONLY USERCODE HAS TO BE SUPPLIED
                BECAUSE SEARCH STOPS ON FIRST ENCOUNTER OF
                GIVEN USERCODE.
         10     CHANGE BY INDEX.  INDEX MUST BE PRESENT.
                PRIORITY CAN BE CHANGED BY SETTING FIELD TO NON-
                ZERO.  CHARGE CAN BE CHANGED BY SETTING CHARGE
                FIELD TO NON-ZERO.  PASSWORD CAN BE CHANGED BY
                SETTING PASSWORD TO NON-ZERO.
         11     CLEAR PACK OVERRIDE FIELD FOR ALL OCCURRENCES OF
                THIS USERCODE.  USERCODE MUST BE SUPPLIED.
         12     CLEAR PACK OVERRIDE BIT FOR ALL OCCURRENCES OF
                THIS USERCODE.  INDEX MUST BE SUPPLIED.

    INDEX       BIT (10)
    USERCODE CHARACTER (10)
                WHEN SET BY PROGRAM (MODE = 0, 2, 4, 5, 7, 8, 9, 11),
                  THE USERCODE MAY OR MAY NOT CONTAIN PARENTHESES.
                  IF PARENS ARE NOT FOUND, ONLY THE FIRST EIGHT
                  USED.
                WHEN SET BY MCP (MODE = 1)
                  USERCODE WILL ALWAYS CONTAIN PARENTHESES.
    PASSWORD    CHARACTER (10)
    PACK NAME   CHARACTER (10)
    CHARGE #    BIT (24)
    PRIORITY    BIT (4)
    PRIVILGD    BIT (1)
    OVERRIDE    BIT (1)
    REINSTATE.MSG.PTR VALUES
                0  NO ERRORS.
                1  ERROR ON INPUT: EITHER INDEX IS WRONG OR
                   USERCODE/PASSWORD IS NOT PRESENT.
                2  "(SYSTEM)/USERCODE" FILE NOT IN "US" SLOT.
```

PROGRAM CALLER

```
CT.VERB     44
48 BITS     SDL DESCRIPTOR
            24 BIT LENGTH OF TEXT
```

24 BIT BASE RELATIVE ADDRESS OF TEXT

This communicate functions in a manner similar to the ZIP communicate described previously, with one notable exception. The program which issues this communicate is suspended, and removed from memory, until the program which is initiated by the communicate goes to end-of-job. This communicate, then, provides a means for one program to call another and wait for its completion. The text which is addressed by the forty-eight bits passed with the communicate should be valid MCP Control Card syntax which causes the execution of a program.

Unfortunately, no means are provided for passing parameters between the two programs involved. This can be done only via the File mechanism of the MCP. The FILE Control Card, described in the Software Operational Guide, does provide some assistance in this area.

The text passed by the communicate must lie within the program's run structure. The program will be automatically discontinued if it does not. No further editing is performed by the communicate. The program which issued the communicate is not informed of the validity of the control syntax passed.

LOAD.DUMP MESSAGE

CT.VERB	46
CT.OBJECT	BASE RELATIVE ADDRESS OF MESSAGE
CT.ADVERB	BIT
	0 1=LOADED 0=DUMPED
	1-11 -

This communicate causes the thirty bytes beginning at the address specified by CT.OBJECT concatenated with either the word "LOADED" or the word "DUMPED", depending upon the setting of bit 1 of the adverb, to be displayed upon the SPO if and only if the LIB system option is set. Refer to the Software Operational Guide for details on the LIB option. All non-blank EBCDIC fields in the message will be shifted to the left before printing until they are separated by no more than one EBCDIC blank.

COMPLEX WAIT (MICRO MCP)

CT.VERB	47
CT.OBJECT	NUMBER OF EVENTS
CT.ADVERB	FIRST EVENT TO CHECK (CHECKED IN CIRCULAR FASHION FROM THIS POINT).
CT.1-ETC.	BIT ENCODED EVENTS (NUMBER SPECIFIED BY CT.OBJECT

```
          MAX=15).
 0- 3 EVENT TYPE
 4- 7 EVENT PARAM1
 8-15 EVENT PARAM2
16-24 EVENT PARAM3
EVENT TYPES:
      0 - NULL - PARAM1,2,3 : NOT USED
      1 - SPO INPUT PRESENT - PARAM1,2,3 : NOT USED
      2 - TIME - PARAM1,2,3 : CONCATENATED BIT 20
          FIELD CONTAINING THE LENGTH OF TIME TO
          WAIT IN 10THS OF A SECOND
      3 - READ OK -PARAM1: NOT USED, PARAM2:
          FILE NUMBER, PARAM3: MEMBER NUMBER IF FILE IS
          Q-FILE-FAMILY
      4 - WRITE OK - PARAM1,2,3: SAME AS READ OK
      5 - QUEUE WRITE OCCURRED - PARAM1: NOT USED,
          PARAM2: FILE NUMBER OF Q-FILE-FAMILY,
          PARAM3: NOT USED
      6 - DATA COMM IO COMPLETE - PARAM1,2,3: NOT USED
REINSTATE.MSG.PTR VALUES
  ZERO RELATIVE INDEX TO THE COMMUNICATE EVENT LIST ELEMENT
  WHICH IS COMPLETE
```

MESSAGE COUNT

```
CT.VERB      48
CT.OBJECT    FILE.NUMBER
CT.ADVERB    0      DECIMAL FORMAT RESULTS IF TRUE
                        COBOL ("PIC 999")
             ELSE BINARY (BIT (24))
             1-11   -
CT.1         RESULT FIELD LENGTH
CT.2         BASE RELATIVE RESULT FIELD ADDRESS
  FUNCTION   RETURN THE COUNT OF THE MESSAGES CONTAINED
             IN THE QUEUE-FILE SPECIFIED. IF THE OBJECT
             IS A QUEUE-FILE-FAMILY, THE COUNT WILL BE
             RETURNED AS A LEFT-JUSTIFIED ARRAY OF
             24-BIT COUNTS, ONE FOR EACH MEMBER OF
             THE FAMILY.
```

RECOVERY COMPLETE

```
CT.VERB      50
```

Refer to P.S. 2212 5470.

GET.ATTRIBUTES

```
CT.VERB      51
CT.OBJECT    FILE NUMBER
```

CT.ADVERB	COMMUNICATE LEVEL (MK 7.0 LEVEL=1)
CT.1	TOTAL ATTRIBUTES (MUST BE 1 IN 7.0)
CT.2	BASE RELATIVE ADDRESS OF ATTRIBUTE LIST

CHANGE.ATTRIBUTES

CT.VERB	52
CT.OBJEXT	FILE NUMBER
CT.ADVERB	COMMUNICATE LEVEL (MK 7.0 LEVEL=1)
CT.1	TOTAL ATTRIBUTES (MUST BE 1 IN 7.0)
CT.2	BASE RELATIVE ADDRESS OF ATTRIBUTE LIST

ACCESS.GLOBALS

CT.VERB	55
CT.OBJECT	0 - CT.3 CONTAINS AN ABSOLUTE MEMORY ADDRESS 1 - HINTS. CT.3 WILL BE USED AS AN OFFSET INTO THE FIELD 2 - RS.NUCLEUS. USE OF CT.ADVERB AND CT.3 IS DESCRIBED BELOW 3 - IOAT. USE OF CT.ADVERB AND CT.3 IS DESCRIBED BELOW 4 - DCH.SCRATCH.MEM 5 - PACK.INFO TABLE 6 - SPO.SQ
CT.ADVERB	SEE BELOW
CT.1 and CT.2	A BASE-RELATIVE SDL DESCRIPTOR WHICH SPECIFIES THE RECEIVING FIELD IN THE PROGRAM. THE FIRST EIGHT BITS OF CT.1 ARE IGNORED BY THE MCP
CT.3	SEE BELOW

Since HINTS actually begins at absolute location zero, there is no functional difference between reading an absolute memory location and reading HINTS with an offset. Both settings of CT.OBJECT are allowed to accomodate possible future expansion to the function of accessing HINTS.

When reading Run Structure Nuclei, all nuclei are returned if CT.ADVERB is set to zero. The number of nuclei that are currently present in memory is returned as a self-relative value in RS.REINSTATE.MSG.PTR. All of the nuclei will be copied to the receiving field in the program in the order that they are linked in memory and up to the limit contained in the size specification in CT.1. If the nuclei of all executing programs are transferred to the receiving field before it is exhausted, the remaining portion of the field will be set to blanks.

To access the Run Structure Nucleus of one particular executing program, CT.ADVERB should be set to one and the job number of the

program should be contained as a twenty-four bit binary value in CT.3. If CT.3 contains zero, the nucleus of the requesting program will be returned. If the nucleus of the program specified by CT.3 is not in memory for any reason, the receiving field will be set to @FFFFFF@ and filled with blanks and a self-relative value of one will be returned in RS.REINSTATE.MSG.PTR.

When reading the IOAT, if CT.ADVERB is set to zero, CT.3 will be used as an offset into the IOAT and the remaining portion of the table will be transferred, up to the limit specified in CT.1. The value of CT.3 may be zero, of course, and the entire table may be transferred. If CT.ADVERB is set to one, CT.3 will be assumed to contain the twenty-four bit binary value of a file number associated with a file which the program currently has open. All of the IOAT entries which follow the associated entry may be transferred, depending upon the value contained in CT.1. If the file is not open or is not present in memory for any reason, @FFFFFF@ will be transferred, the remainder of the receiving field will be set to blanks and RS.REINSTATE.MSG.PTR will be set to a self-relative value of one.

If CT.ADVERB is set to a value of two when reading the IOAT, the low-order twelve bits of CT.3 will be assumed to contain a Port/Channel/Unit combination in the following format:

Bits 12 - 14	Port
Bits 15 - 18	Channel
Bits 19 - 23	Unit

The MCP will scan the IOAT for the entry associated with the specified unit and will return that entry plus all subsequent entries up to the limit of the IOAT or that specified in CT.1. If the specified unit is not present in the IOAT, the MCP will set the receiving field to @FFFFFF@ followed by blanks and will set RS.REINSTATE.MSG.PTR to a self-relative value of one.

INDEXED SEQUENTIAL POSITION

CT.VERB	56	
CT.OBJECT	FILE NUMBER	
CT.ADVERB	BIT	
	0	-
	1	REPORT TO USER ON PARITY
	2	-
	3	RESULT MASK FIELDS PRESENT
	4-5	-
	6-7	RELATIONAL OPERATOR
		0 EQUAL TO
		1 GREATER THAN

```
                    2       NOT LES THAN (> I =)
                8-10    SELECTION CONDITION
                        0       NEXT
                        1       PRIOR
                        2       FIRST
                        3       LAST
                        4       NEXT AT
                        5       CURRENT
                        6       AT
                        7       RANDOM
                11      -
CT.1            LENGTH OF RESULT MASK
CT.2            ADDRESS OF RESULT MASK
CT.3            -
CT.4            -
CT.5            STRUCTURE NUMBER
CT.6            KEY ADDRESS
CT.7            KEY LENGTH
```

INDEXED SEQUENTIAL READ

```
CT.VERB         57
CT.OBJECT       FILE NUMBER
CT.ADVERB       BIT
                0       REPORT TO USER ON EOF
                1       REPORT TO USER ON PARITY
                2       -
                3       RESULT MASK FIELDS PRESENT
                4-5     -
                6-7     RELATIONAL OPERATOR
                        0       EQUAL TO
                        1       GREATER THAN
                        2       NOT LESS THAN (> I =)
                8-10    SELECTION CONDITION
                        0       NEXT
                        1       PRIOR
                        2       FIRST
                        3       LAST
                        4       NEXT AT
                        5       CURRENT
                        6       AT
                        7       RANDOM
                11      -
CT.1            LENGTH OF RESULT MASK
CT.2            ADDRESS OF RESULT MASK
CT.3            LOGICAL RECORD LENGTH
CT.4            LOGICAL RECORD ADDRESS
CT.5            STRUCTURE NUMBER
CT.6            KEY ADDRESS
CT.7            -
```

INDEXED SEQUENTIAL WRITE

CT.VERB	58
CT.OBJECT	FILE NUMBER
CT.ADVERB	BIT
	0 REPORT TO USER ON EOF
	1 REPORT TO USER ON PARITY
	2 -
	3 RESULT MASK FIELDS PRESENT
	4-11 -
CT.1	LENGTH OF RESULT MASK
CT.2	ADDRESS OF RESULT MASK
CT.3	LOGICAL RECORD LENGTH
CT.4	LOGICAL RECORD ADDRESS
CT.5	STRUCTURE NUMBER
CT.6	KEY ADDRESS
CT.7	-

INDEXED SEQUENTIAL REWRITE

CT.VERB	59
CT.OBJECT	FILE NUMBER
CT.ADVERB	BIT
	0 -
	1 REPORT TO USER ON PARITY
	2 -
	3 RESULT MASK FIELDS PRESENT
	4-11 -
CT.1	LENGTH OF RESULT MASK
CT.2	ADDRESS OF RESULT MASK
CT.3	LOGICAL RECORD LENGTH
CT.4	LOGICAL RECORD ADDRESS
CT.5	STRUCTURE NUMBER
CT.6	KEY ADDRESS
CT.7	-

INDEXED SEQUENTIAL DELETE

CT.VERB	60
CT.OBJECT	FILE NUMBER
CT.ADVERB	BIT
	0 -
	1 REPORT TO USER ON PARITY
	2 -
	3 RESULT MASK FIELDS PRESENT
	4-11 -
CT.1	LENGTH OF RESULT MASK
CT.2	ADDRESS OF RESULT MASK
CT.3	-
CT.4	-
CT.5	STRUCTURE NUMBER
CT.6	KEY ADDRESS

```
CT.7            -
```

RELATIVE I/O COMMUNICATE = START

```
CT.VERB         61
CT.OBJECT       FILE NUMBER
CT.ADVERB       BIT
                0       REPORT TO USER ON EOF
                1       REPORT AND RETURN TO USER ON PARITY
                2       REPORT AND RETURN TO USER (INCOMPLETE I/O)
                3       RESULT MASK FIELD PRESENT
                4-5     -
                6-7     RELATIONAL OPERATOR
                        0       EQUAL TO
                        1       GREATER THAN
                        2       NOT LESS THAN
                8-11    -
CT.1            LOGICAL RECORD BIT LENGTH
CT.2            LOGICAL RECORD BASE RELATIVE BIT ADDRESS
CT.3            ACTUAL BINARY DISK KEY (RELATIVE KEY)
                SUPPLIED BY USER
CT.4            -
CT.5            LENGTH IN BITS OF RESULT MASK FIELD
CT.6            BASE RELATIVE ADDRESS OF RESULT MASK FIELD
REINSTATE.MSG.PTR
                0       GOOD READ
                1       END OF FILE
                2       I/O ERROR
                3       INCOMPLETE I/O
   (ADDITIONAL ITEMS FOR FILE STATUS DEFINED IN THE SEQUENTIAL
    FILES DESIGN SPECIFICATION)
```

RELATIVE I/O COMMUNICATE = WRITE

```
CT.VERB         62
CT.OBJECT       FILE NUMBER
CT.ADVERB       BIT
                0       REPORT TO USER ON EOF
                1       REPORT AND RETURN TO USER ON PARITY
                2       REPORT AND RETURN TO USER (INCOMPLETE I/O)
                3       RESULT MASK FIELD PRESENT
                4       ACCESS TYPE
                        0  SEQUENTIAL (NEXT)
                        1  RANDOM (AT KEY)
                5-11    -
CT.1            LOGICAL RECORD BIT LENGTH
CT.2            LOGICAL RECORD BASE RELATIVE BIT ADDRESS
CT.3            ACTUAL BINARY DISK KEY FOR RANDOM OR DYNAMIC
                FILES (SUPPLIED BY USER; NOTHING IF IN
                SEQUENTIAL MODE)
CT.4            -
CT.5            LENGTH IN BITS OF RESULT MASK FIELD
```

```
CT.6            BASE RELATIVE ADDRESS OF RESULT MASK FIELD
REINSTATE.MSG.PTR
                0       GOOD READ
                1       END OF FILE
                2       I/O ERROR
                3       INCOMPLETE I/O
   (ADDITIONAL ITEMS FOR FILE STATUS DEFINED IN THE SEQUENTIAL
    FILES DESIGN SPECIFICATION)
```

RELATIVE I/O COMMUNICATE = REWRITE

```
CT.VERB         63
CT.OBJECT       FILE NUMBER
CT.ADVERB       BIT
                0       REPORT TO USER ON EOF
                1       REPORT AND RETURN TO USER ON PARITY
                2       REPORT AND RETURN TO USER (INCOMPLETE I/O)
                3       RESULT MASK FIELD PRESENT
                4       ACCESS TYPE
                        0  SEQUENTIAL (NEXT)
                        1  RANDOM (AT KEY)
                5-11    -
CT.1            LOGICAL RECORD BIT LENGTH
CT.2            LOGICAL RECORD BASE RELATIVE BIT ADDRESS
CT.3            ACTUAL BINARY DISK KEY FOR RANDOM OR DYNAMIC
                FILES (SUPPLIED BY USER; NOTHING IF IN
                SEQUENTIAL MODE)
CT.4            -
CT.5            LENGTH IN BITS OF RESULT MASK FIELD
CT.6            BASE RELATIVE ADDRESS OF RESULT MASK FIELD
REINSTATE.MSG.PTR
                0       GOOD READ
                1       END OF FILE
                2       I/O ERROR
                3       INCOMPLETE I/O
   (ADDITIONAL ITEMS FOR FILE STATUS DEFINED IN THE SEQUENTIAL
    FILES DESIGN SPECIFICATION)
```

THE REWRITE COMMUNICATE WILL BE ESSENTIALLY THE SAME AS THE WRITE, BUT WILL HAVE A DISTINCT MEANING IN LOGICAL I/O

RELATIVE I/O COMMUNICATE = DELETE

```
CT.VERB         64
CT.OBJECT       FILE NUMBER
CT.ADVERB       BIT
                0       REPORT TO USER ON EOF
                1       REPORT AND RETURN TO USER ON PARITY
                2       REPORT AND RETURN TO USER (INCOMPLETE I/O)
                3       RESULT MASK FIELD PRESENT
                4       ACCESS TYPE
                        0  SEQUENTIAL (NEXT)
```

```
                    1  RANDOM (AT KEY)
               5-11 -
CT.1           -
CT.2           -
CT.3           ACTUAL BINARY DISK KEY FOR RANDOM OR DYNAMIC
               FILES (SUPPLIED BY USER; NOTHING IF IN
               SEQUENTIAL MODE)
CT.4           -
CT.5           LENGTH IN BITS OF RESULT MASK FIELD
CT.6           BASE RELATIVE ADDRESS OF RESULT MASK FIELD
REINSTATE.MSG.PTR
               0     GOOD READ
               1     END OF FILE
               2     I/O ERROR
               3     INCOMPLETE I/O
   (ADDITIONAL ITEMS FOR FILE STATUS DEFINED IN THE SEQUENTIAL
    FILES DESIGN SPECIFICATION)
```

RELATIVE I/O COMMUNICATE = READ

```
CT.VERB        65
CT.OBJECT      FILE NUMBER
CT.ADVERB      BIT
               0     REPORT TO USER ON EOF
               1     REPORT AND RETURN TO USER ON PARITY
               2     REPORT AND RETURN TO USER (INCOMPLETE I/O)
               3     RESULT MASK FIELD PRESENT
               4     ACCESS TYPE
                     0  SEQUENTIAL (NEXT)
                     1  RANDOM (AT KEY)
               5-11  -
CT.1           LOGICAL RECORD BIT LENGTH
CT.2           LOGICAL RECORD BASE RELATIVE BIT ADDRESS
CT.3           ACTUAL BINARY DISK KEY FOR RANDOM OR DYNAMIC
               FILES (SUPPLIED BY USER; NOTHING IF IN
               SEQUENTIAL MODE)
CT.4           -
CT.5           LENGTH IN BITS OF RESULT MASK FIELD
CT.6           BASE RELATIVE ADDRESS OF RESULT MASK FIELD
REINSTATE.MSG.PTR
               0     GOOD READ
               1     END OF FILE
               2     I/O ERROR
               3     INCOMPLETE I/O
   (ADDITIONAL ITEMS FOR FILE STATUS DEFINED IN THE SEQUENTIAL
    FILES DESIGN SPECIFICATION)
```

SEQUENTIAL REWRITE (MMCP)

```
CT.VERB        66
CT.OBJECT      FILE NUMBER
CT.ADVERB      BIT
```

	0 REPORT AND RETURN TO USER ON EOF
	1 REPORT AND RETURN TO USER ON PARITY
	2 REPORT AND RETURN TO USER ON INCOMPLETE I/O
	3 LENGTH ADDRESS PART IS PRESENT FOR THE RESULT MASK
	4-11 -
CT.1	LOGICAL RECORD BIT LENGTH
CT.2	LOGICAL RECORD BASE RELATIVE BIT ADDRESS
CT.3	RANDOM FILE ACTUAL BINARY KEY
CT.4	-
CT.5	LENGTH IN BITS OF RESULT MASK
CT.6	BASE RELATIVE ADDRESS OF RESULT MASK FIELD

INDEXED/SEQUENTIAL OPEN

CT.VERB	67
CT.OBJECT	FILE NUMBER
CT.ADVERB	BIT
	0 REPORT.FILE.MISSING
	1 REPORT.FILE.LOCKED
	2 REPORT.EXCEPTION (SECURITY ERRORS)
	3-11 -
	(THE OPEN TYPE IS TAKEN FROM THE FPB.ADVERB AND FPB.EXPANDED.ADVERB FIELDS)
CT.1	LENGTH OF USERCODE/PASSWORD FIELD (IF OPEN.ON.BEHALF.OF)
CT.2	BASE RELATIVE ADDRESS OF USERCODE/PASSWORD FIELD
CT.3	OPEN STATUS - RESERVED FOR THE SMCP TO KEEP TRACK OF WHERE TO RESUME IF THE ENTIRE OPEN CANNOT BE COMPLETED.

INTER-PROCESS COMMUNICATION

QUEUE SYSTEM AND INTERFACES

A message queue system has existed in MCP II since 1973. This section describes the current Queue implementation and the interfaces between the Queue system and other system software.

The word "Queue" as used in this document, most often refers to the actual data structure maintained by the operating system. This data structure is used as a means of inter-process communication. Queues may have various attributes just as files do. For example, Queues may have two ten-character names, user counts, message counts, and so forth. The data structure is used to address a list of messages. This list may be empty. A Queue user may add to the back or remove from the front of this list. The Queue may be shared -- one or more processes may put messages in the list and one or more processes may remove messages. Only the MCP may access the data structure directly. User programs must use other mechanisms, which are constructed from this data structure, such as Queue Files or Remote Files.

DESIGN PHILOSOPHY

The design of the data structure (Figure 8-1) was strongly affected by the need to reduce the S-memory needs of queues. Reusable structures like message buffers and message descriptors are pooled for the use of the whole queue system. The memory space used by empty message buffers and descriptors is not automatically returned to the system. The Queue implementation, rather, retains them for later use. This results in quicker allocation when this space is required again and in less disturbance of the working set of the code in the system. Since Queue Files and Remote Files are unblocked, their FIB's need not have buffers. This minimizes the amount of memory required to contain the FIB.

QUEUE FILE FAMILIES

A Queue File Family may consist of a maximum of 1023 Queues, each having the same first name or MFID and the same attributes. A Queue File Family is shown diagrammatically in Figure 8-2. A user program may READ a message from one of the individual Queues in the family or it may request a message from any queue in the family. On a WRITE operation, however, the individual family member must be specified.

If individual Queues of a Queue File Family are to be addressed, the individual member must be specified by an ordinal number, or key, much like Switch Files in certain languages. A key of zero specified on a Read of a Queue File Family means that the user will accept a message from any of the individual Queues which comprise the family.

QUEUE DESCRIPTORS

For a given Queue, the Queue name, maximum length, pointers to first and last messages, etc, are stored in the Queue Descriptor. The descriptor must be in memory during the existence of the Queue. Users of the Queue are given "Q-keys", which serve as pointers to the Queue Descriptor, when the Queue File is opened and the user has specified the desired attributes of the Queue. For a Queue File, the Q-key is stored in the file's FIB. If the Queue is empty, the 360-bit descriptor is the only memory structure dedicated solely to the Queue.

QUEUE DISK

Messages stored in a queue may reside on disk or in memory. At Queue creation, an area of system disk is obtained for the Queue large enough to hold Q.MAX.MESSAGES of size Q.MAX.MESSAGE.SIZE. These two attributes are normally specified by the user. A Queue specified to contain a maximum of 255 messages, each of a maximum size of 200 bytes will require 255 *((200+179) DIV 180) or 510 disk segments, where DIV denotes an Integer Divide operation. The required disk space will be allocated when the Queue is opened, prior to the time it is actually required. This is done to minimize the processing required to store the message on disk. Users who have minimal amounts of disk storage available may control the amount that is required by Queues by manipulating Q.MAX.MESSAGES. The disk space that is allocated to Queues is not locked in the directory. If the system fails while Queues are active, the disk is returned to the available list during the ensuing Clear/Start operation. Disk is always allocated to Queues, even if sufficient memory is available to contain the maximum number of messages.

Queue messages are written to disk when a message being put into a queue makes the count of messages in memory equal to the attribute Q.BUFFERS. When this situation occurs, a disk Write operation on the last message in memory in the Queue is initiated. This will make one of the buffers available for an ensuing insertion in the Queue. There is one exception to this statement. The disk Write operation will not be initiated by the Queue routines if the attribute Q.BUFFERS is equal to or greater than the attribute Q.MAX.MESSAGES. In this case, messages

associated with the Queue may only be written to disk by the Memory Management routines. The Memory Management routines may write Queue messages to disk anytime memory is required in the system. This ensures that Queue messages will not fill memory to the point where thrashing occurs.

When a user removes, or READs, a message from a Queue the first message in the Queue is transferred to his Run Structure and the next message in the Queue is examined to determine if it is on disk. If it is, a look-ahead disk Read operation is initiated to minimize the time that the user will have to wait for delivery of the next Queue message.

The I/O descriptors that are used for the disk Read and Write operations just described reside in the Queue File's FIB. For each mode of use, input or output, a program opening a Queue File is given one I/O descriptor. A file opened input and output is given two I/O descriptors. I/O descriptors are shared among all members of a Queue File Family, so that no Queue File FIB will ever contain more than two I/O descriptors.

MESSAGE DESCRIPTORS

The method of storing messages in the queue is by means of a linked list of Message Descriptors. Each Message Descriptor (MD) consists of an 80-bit system descriptor and two link fields, for a total of 128 bits each. The system descriptor actually describes the message text, according to normal MCP conventions.

To reduce checkerboarding, MD's are allocated in blocks of ten. Assigning a Message Descriptor to a message is accomplished by searching the block(s) of ten for an available MD. If none are available, memory space for an additional block of ten is obtained via a call on the Memroy Management routines. The blocks of Message Descriptors are surveyed periodically to consolidate and return unused blocks to the system. At least one block is retained as long as any Queues exist.

MESSAGE BUFFERS

If a queued message is in memory, the memory area which contains the message is known as a Message Buffer (MB). The ML.TYPE field in the memory link which describes the area will be set to a unique value which denotes a Message Buffer. No Queue will ever have more than Q.BUFFERS messages in memory at any time, including those messages which are in transit between memory and disk. Actually, since the Memory Management routines are capable

of writing Queue messages to disk and removing them from memory, the Queue routines cannot guarantee that any messages will be in memory at any given time.

QUEUE ATTRIBUTES

In addition to attributes common to all files, the user may specify two attributes whose interpretation has meaning for Queue Files only:

1. Q.MAX.MESSAGES - the maximum number of messages a Queue can store, at which point it is considered full (maximum 1023).

2. Q.FAMILY.SIZE - the number of sub-queues in a Queue File Family (maximum 1023).

In addition, Q.BUFFERS as described in the foregoing may be specified by the BUFFERS file attribute. Thus, the user may have some control over the number of messages that may be contained in memory at any given time. In the COBOL Language, a Queue File Declaration may appear as:

```
SELECT MY.Q  ASSIGN TO QUEUE.
  .
  .
  .
FD MY.Q VALUE OF Q.MAX.MESSAGES IS 20
  RESERVE 3 ALTERNATE AREAS.
01 MY.Q.BUF     PIC X(80).

SELECT MY.QFF  ASSIGN TO QUEUE.
  .
  .
  .
FD MY.QFF  FILE CONTAINS 3 QUEUES
  VALUE OF Q.MAX.MESSAGES IS 10
  RESERVE 2 ALTERNATE AREAS.
01 MY.QFF.BUF  PIC X(80).
```

If a Queue File Family is opened, the same attributes apply to every member individually. In MY.QFF above, for example, all three members may hold ten messages, each having a maximum of two in memory.

The name assigned to a Queue File is specifed by the user as the MFID/FID combination. For a Queue File Family, the MFID is

specified by the user and is taken to be the first ten characters, and an FID is synthesized from the member number for each queue in the family. The first member of MY.QFF would be named "MY.QFF/#00000001".

When a Queue File is opened, the MCP compares the 20-character name with the names of all Queues currently in existence. If a Queue of that name if found, the opener is linked to the existing queue and the Queue's user count is incremented. If the Queue does not exist, a new Queue is created with the attributes provided by the FPB. Queue attribute binding occurs when the Queue is first created, by the first process to open the Queue File. If two programs share a Queue (e.g., both agree on the name), the first program to open the shared Queue file binds the attributes.

Blocking of records is not allowed in Queue Files. The Record Size attribute determines the upper limit on the length of a message which may be stored in a queue file.

QUEUE FILE LOGICAL I/O OPERATIONS

Queue File logical I/O operations are rather simple and straightforward. As mentioned previously, all Queue Files must be unblocked. Truncation or blank fill may occur, depending upon the size of the user's work area and the size of the message being moved, exactly as is done on all other logical I/O operations on B1000 systems. The user may request that three different exception conditions be reported to him on all Queue File logical I/O operations. These three conditions are, in COBOL syntax:

1. ON END-OF-FILE
2. ON EXCEPTION, and
3. ON INCOMPLETE-IO.

On READ operations, END-OF-FILE is reported to the user when the Queue File is empty and no program exists which has the Queue opened for output. EXCEPTION is reported on Queue File Families only and on READ operations only, and actually denotes an invalid key passed on the READ to specify the desired family member. INCOMPLETE-IO is reported if the Queue is empty but programs still exist which have the Queue opened for output purposes. All three conditions are reported only if requested, of course. Failure to request that END-OF-FILE or EXCEPTION be reported will be considered a program error if either condition occurs and the program will be discontinued.

The precise meaning of EOF on READ is that (a) the last writer on this Queue has closed his Queue File and (b) the Queue is empty. EOF is treated as a pseudo-message in the Queue. That is, when the last message has been read from the Queue File, the File still exists and actually remains "not empty" for WAIT purposes. A subsequent READ will result in the EOF branch being taken. The Queue is then empty, but still in EOF status, so if yet another READ is issued on the Queue File, the reader will again take the EOF branch. EOF can be cleared by either the reader closing and reopening the file or by the opening of the Queue by a new writer.

READs to specific members of a Queue File Family are treated exactly like READs on single Queue Files. An unspecific READ on a Queue File Family will return EOF only if all members of the Family are at EOF (i.e., empty, no writers). When the last writer closes any member queue of a QFF, the event Q.WRITE.OCCURRED will be caused for the QFF; this will put a reader in the READY.Q when it WAITS on this event.

A MESSAGE COUNT communicate operator is implemented to enable user programs to determine if any messages are present in the Queue Files they are using. This function is described in a later paragraph. A MESSAGE COUNT communicate issued for a Queue that is marked as being at END-OF-FILE will show the EOF status as a pseudo-message - the count for that particular Queue File will be one more than the count of real messages. When the reader executes a specific READ on the member Queue which is at EOF, the EOF branch will be taken. The next MESSAGE.COUNT will show the member Queue as containing no messages. Another READ on the member will result in the EOF branch being taken again, as is done for a single Queue File.

On Queue WRITE operations, END-OF-FILE is not defined and will never occur. EXCEPTION has the same meaning as it does for READ operations - it denotes an invalid key condition on Queue File Families only. INCOMPLETE-IO will be reported, if requested, when the Queue is full and there is no space available to store the message that is being written. If no INCOMPLETE-IO report is rquested and the Queue is full when a WRITE occurs, the program is suspended until space is available in the Queue.

As mentioned previously, when a logical I/O request is directed to a Queue File Family, a key must be included to identify the specific Queue in the family to which the operation is directed. This is similar to Switch Files in the SDL Language. Family members are numbered logically from one to n. Giving a key of zero on a READ is defined as an unspecified read. The members

will be searched, beginning with number one, and the first queue member found not empty will be read. A key of zero on a write is invalid.

WRITING TO THE TOP OF A QUEUE FILE

Writing to the top of a Queue File is allowed in the MCP though it may or may not be allowed in a given language. A message written to a queue file normally goes to the bottom of the Queue though some rare occurrences in applications may require the converse. This capability is invoked in the communicate operation by setting bit 7 of CT.ADVERB.

MESSAGE.COUNT COMMUNICATE

This communicate operation returns the count of messages in the Queue File specified. If a Queue File Family is specified, the count of each member will be returned in an array (member one in the first position, member two in the second, etc.), up to the limit of the result field. Counts will be returned either in decimal (COBOL "PICTURE 999") or binary (SDL "BIT(24)") depending on the value of the first bit of CT.ADVERB. This operation may not be implemented in all languages.

Format:

CT.VERB	48 (HEX 0030)
CT.OBJECT	File Number
CT.ADVERB BIT 1	Decimal format results if true
CT.1	Result field length in bits
CT.2	Result field address

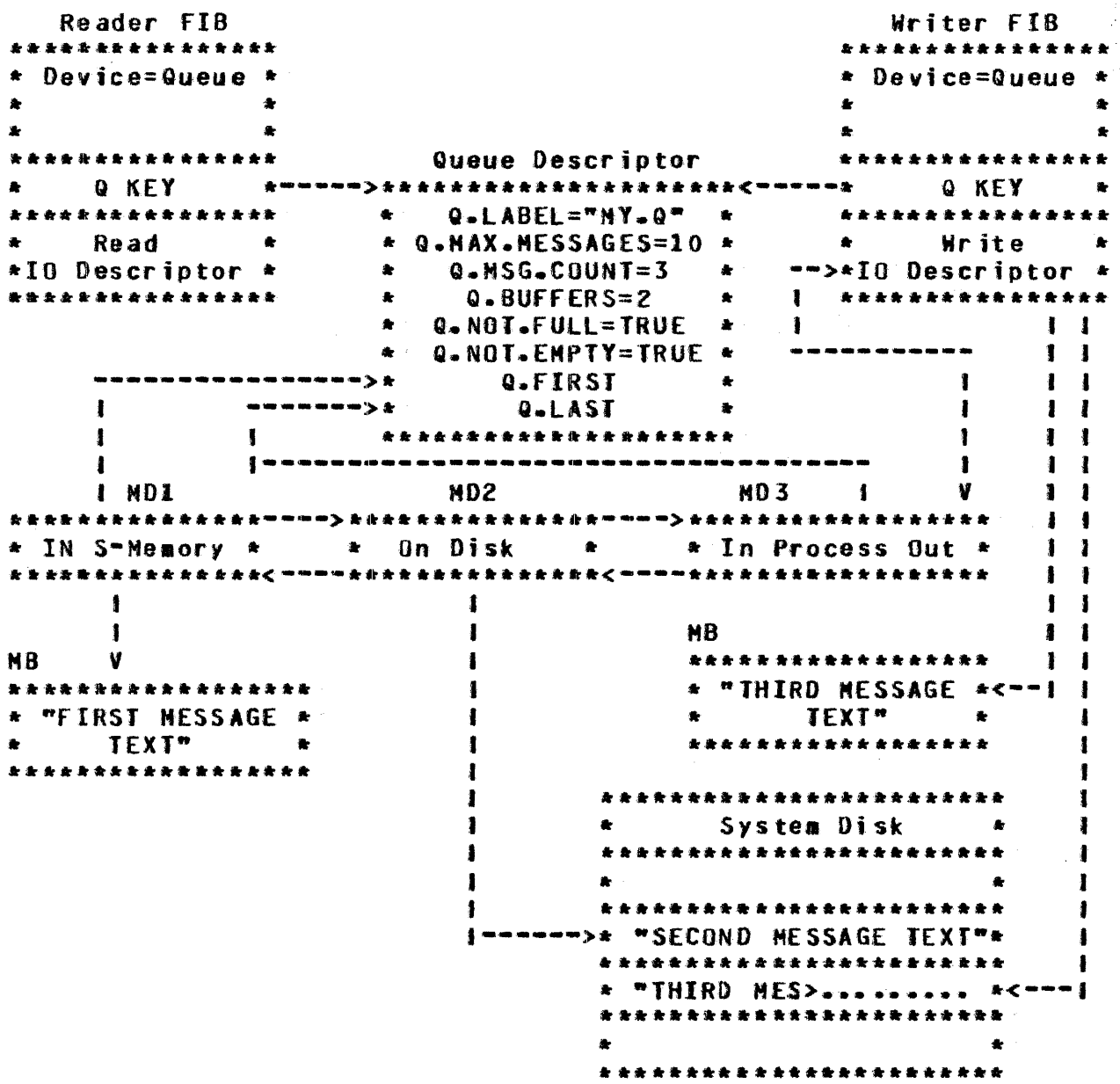


Figure 8-1: Two Programs Communicating in a Queue File Called "MY.Q". The Queue contains three messages.

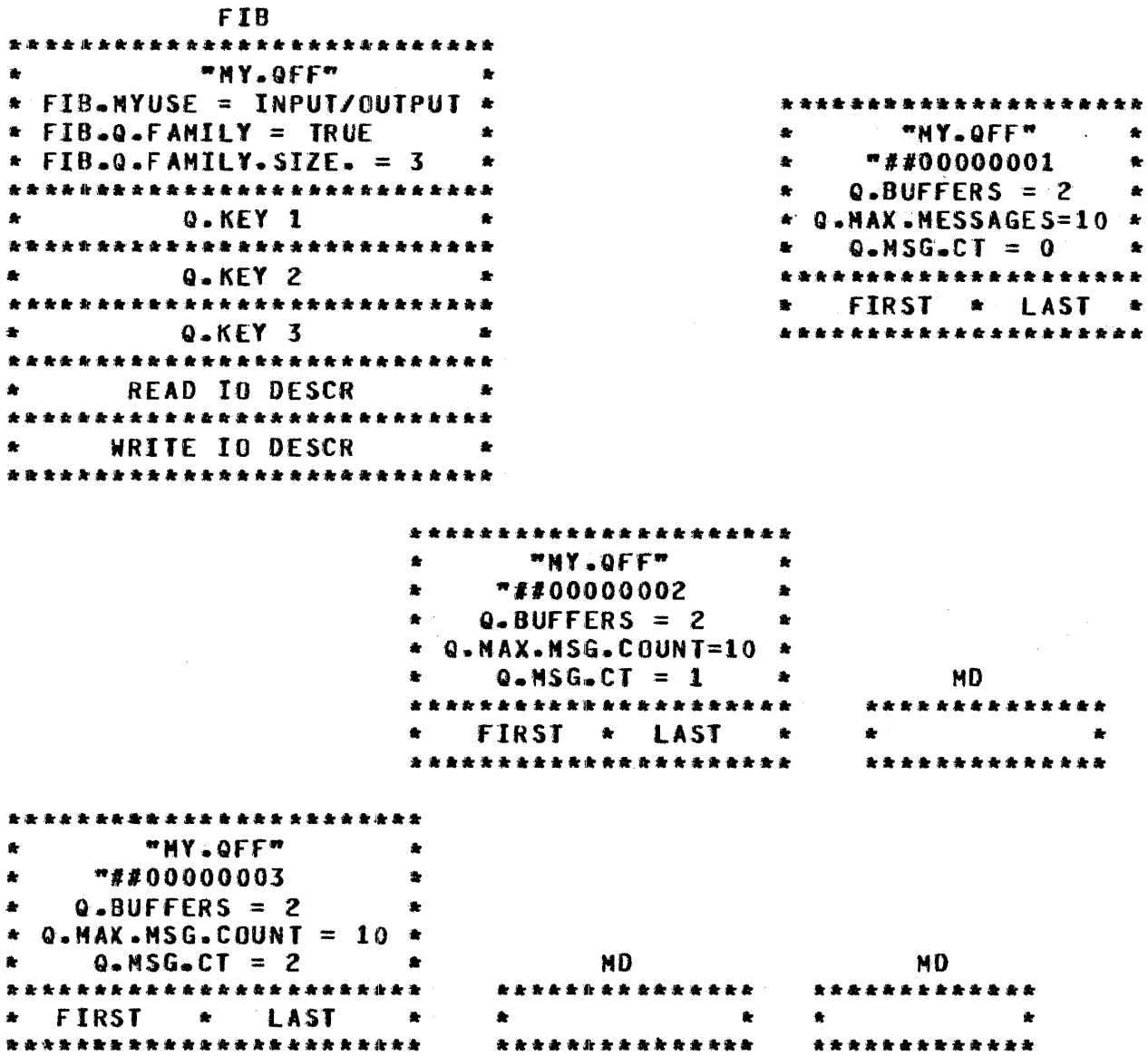


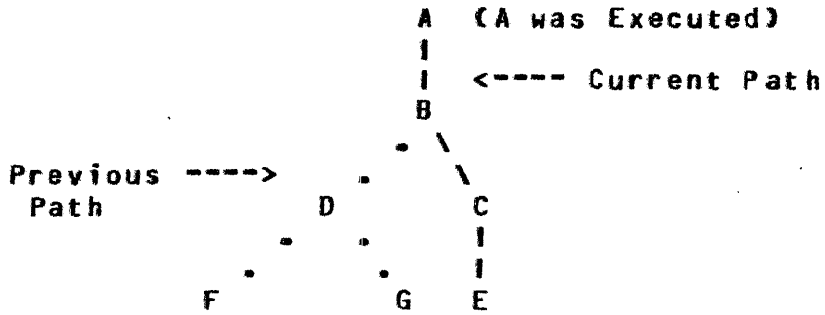
Figure 8-2: A Queue-file Family with three members.

INTER-PROGRAM COMMUNICATION

Another means of accomplishing inter-process communication is the Inter-Program Communication Module, first implemented in the 9.0 software to satisfy the requirements of the ANSI '74 COBOL Language. According to the specifications of that language, the facility provides synchronous CALL and EXIT verbs, as well as a shared data implementation. The module provides a facility to transfer control from one program to another and the ability for both programs to have access to the same data items. The names of the programs to which control is to be passsed may or may not be known at compile time. Additionally, this module provides the ability to determine the availability of memory for the program to which control is being passed.

RUN UNIT DEFINITION

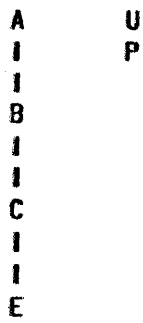
The definition of a "Run Unit" is critical to the implementation of the CALL/CANCEL mechanism described in the ANSI '74 COBOL specifications. The execution of any program via an EXECUTE control instruction does not establish a Run Unit. A Run Unit is established only when an executed program initiates another program via the CALL communicate. That called program is now a member of the Run Unit associated with the program that was originally executed. Similarly, any program called by a program within the Run Unit becomes part of that Run Unit and remains in that Run Unit until terminated or cancelled. A job cannot be a member of more than one run unit. The following figure represents seven programs (A - G) which have been called within a run unit.



The connecting links are generated by and represent the last used path, and the link exists until a return (EXIT PROGRAM) is accomplished. Once a called program has been exited (D, F, G), it remains suspended in its current state. The only path that is of interest is the path last traversed.

The current path is important in order to check the validity of a CALL or CANCEL statement; if a program tries to CALL or CANCEL itself or any of its predecessors, the entire run unit will be DS'ed. The other links are unimportant, as any program in the run unit can CALL or delete other existing programs, with the previously mentioned exceptions, or can CALL new programs.

If, for example, program 'E' cancels program 'D' then the Run Unit would consist of all of the following programs and appears as:



A CALL to any of these programs will result in a transfer of control to the existing state, whereas a CALL to any other program, including 'D', will cause an initial state copy to be invoked before control is transferred. The termination, via STOP RUN or ABORT, of any program in a Run Unit will result in the removal of all programs in that Run Unit from memory.

IPC IMPLEMENTATION OF SHARED DATA

For those not familiar with the ANSI '74 COBOL definition of Inter-Program Communication, all programs within a Run Unit execute synchronously. No two programs in a Run Unit may be executing simultaneously at any time and consequently, there are no problems associated with two or more programs contending for the use of shared data. Control is passed to a program via the CALL verb. The program which contains the CALL will not be allowed to execute again until the called program performs an EXIT PROGRAM verb.

The calling program may specify one or more data items to which the called program has access. The shared data may be any 01 or 77 level item described in the calling program This includes items whose addresses have been received through a CALL. The data items may be named and defined differently in each program as long as the length of the item remains the same in each program. This mechanism is stricly a "pass by name" facility. Parameters cannot be passed by value. Additionally, storage for

the shared data is never allocated in the called program. In other words, the address of the data, only, is always passed to the called program.

IPC RUN STRUCTURE NUCLEUS CHANGES

In order to maintain all of the necessary information regarding the programs which comprise a Run Unit, several fields were added to the Run Structure Nucleus, RS.NUCLEUS, the field in memory which contains information about each program that is executing, in the 9.0 version of the MCP. This field, as it has always been, is shared by the Operating System and the user program's interpreter. The following is a list of the fields which were added in the 9.0 version and a brief description of each.

RS.RUN.UNIT BIT(16)

When a job initiates a CALL, he establishes a RUN UNIT. This Run Unit is identified by his own (the originator's) job number. RS.RUN.UNIT, for any job in the Run Unit, will contain the job number of the program which initiated the Run Unit.

RS.RUN.UNIT.LINK BIT(16)

This field will contain zero for the job that initiated the run unit and for any job in the Run Unit that has done an EXIT PROGRAM. For any job that is currently active in the Run Unit, a job that has not done an EXIT, this field will contain the job number of his caller.

RS.IPC.DICT BIT(24)

This field will contain the absolute address of the IPC.DICTIONARY through which parameters will be accessed within the calling job's base-limit space. The field will be zero if the dictionary does not exist. This is the list of parameters that this job will pass. The IPC.DICTIONARY is adjacent to the RS.NUCLEUS and found only in the callers Run Structure Nucleus.

RS.IPC.PARAMETER.LIST BIT(24)

This field will contain the absolute address of the IPC.PARAMETER.LIST. This space will be adjacent to the Run Structure Nucleus for any called job that can receive parameters. The IPC.PARAMETER.LIST will be a series of 24 bit fields. The first field will contain the number of parameters that this job is capable of receiving. The remaining fields in the list will contain the length in bits of each parameter. This list is built only for the called program from the IPC.PARAMETER.LIST in the called program's code file that is generated by the compiler. If the job cannot receive parameters, this field will contain zero.

RS.IPC.DICT.SIZE BIT(16)

This field will contain the number of entries in this program's IPC Dictionary.

RS.EXECUTE.TYPE BIT(4)

This field is used to store the type of execution that originated the job. If the job is not an Execute type or a Call type, then it cannot be called. The field can contain the following values:

1 = Execute
2 = Compile and Go
3 = Compile for Syntax
4 = Compile to Library
5 = Compile and Save
6 = Go Part of Compile and Go
7 = Go Part of Compile and Save
8 = Call

RS.NAME CHARACTER(30)

This field will contain the name of this program. In the case of compilations, denoted by he value of the previous field, it will contain the name of the compiler as well.

RS.CALLERS.LR BIT(24)

This field will contain the Limit Register of this job's caller.

RS.IPC.EVENT BIT (1)

This field is a dummy event for any IPC hang or suspension of execution. If a program is waiting on RS.IPC.EVENT and is currently passive, which will be indicated by a zero value in RS.RUN.UNIT.LINK, the RS.STATUS will be set to a value to indicate "Waiting to be Called". If the program is currently active, indicated by a non-zero value in RS.RUN.UNIT.LINK, the RS.STATUS will be set to a value indicating "Waiting on called program".

RS.CANCELED BIT(1)

If this boolean is true, then at least one CANCEL communicate has been issued against this program. When this is true, this particular job is effectively no longer a member of the Run Unit and is waiting to be terminated by the SMCP.

IPC Program Parameter Block Changes

It was also necessary to make changes in the Program Parameter Block, the two-sector field that is generated by the compilers and stored in the code file, to accomodate the IPC implementation. A list of the fields that have been added is presented below.

PROG.IPC.SIZE BIT(16)

This field indicates the number of entries in the IPC.PARAMETER.LIST. If this field is not equal to zero, it indicates to the MCP that this program can only be called - it can never be EXECUTEd.

PROG.IPC.PIR BII(24)

This field is used to store the relative disk address in the code file of the IPC.PARAMETER.LIST. The IPC.PARAMETER.LIST will be a series of 24 bit fields that contain the length in bits of the parameters that may be passed to this program with a CALL.

PROG.IPC.MAX.SEND.PARAMS BII(16)

This field indicates the maximum number of parameters to be passed by this program through a CALL, which will also be the number of entries in the IPC Dictionary.

It was also necessary to add a field to the format of the Program Parameter Block that is used by the MCP after the job is scheduled for execution. This field, known as PPB.RUN.UNIT, is sixteen bits in length and is used to contain the Job number of the run unit that this program will become a part of.

IPC.DICTIONARY

The IPC.DICTIONARY is a list of System Descriptors built by the program to describe the parameters to be passed on a CALL. This dictionary will be within the space defined by RS.IPC.DICT in the RS.NUCLEUS of the calling program. The length of this dictionary is passed in the CALL communicate. The Micro MCP will verify that the number and length of parameters passed match the IPC.PARAMETER.LIST of the called program.

IPC COMMUNICATE OPERATOR

One new communicate operator was added to the Operating System to accomodate the IPC implementation. This operator is generated by the Compilers to implement the CALL, CANCEL and EXIT PROGRAM verbs. It may be handled by the Micro MCP or the SDL MCP, depending upon the circumstances. Its format is presented below and in the Demand Management section.

CT.VERB	43
CT.OBJECT	0 = CALL 1 = CANCEL 2 = EXIT PROGRAM (No EOJ)
CT.ADVERB	Bit 0 - if CALL, return on NO MEMORY 1-11 - Not used
CT.1	Base relative address of a 30 character field that contains the name of the job to be called or cancelled.
CT.2	Number of parameters to be passed

RS.REINSTATE.MSG.PTR values returned if requested:

0 - Communicate completed as requested.
1 - For CALL, insufficient memory to complete the CALL.
- For EXIT PROGRAM, the program was initiated by an EXECUTE instruction as opposed to a CALL.
- Not used for CANCEL.

IPC Verb Operation

One of the primary objectives of the IPC implementation was performance. Therefore, as much as possible of the IPC function was implemented in the Micro MCP. In the ANSI '74 COBOL Language, the CALL and CANCEL verbs require the specification of program names within the source text. On the B1000 system, the name of a program may be unknown to the user when the program is compiled, since the Run Unit may be executed under a Usercode.

To simplify the task of associating program names with those specified by a program in a CALL or CANCEL communicate, a new system structure was implemented. A programmatic description of the structure, IPC.RUN.UNIT.LIST, is presented below

```
01 IPC.RUN.UNIT.LIST              BIT(320),

      02 IPC.RUN.UNIT.NUMBER        BIT (16),
      02 IPC.PGM.NAME               CHAR(30),
      02 IPC.PGM.JOB.NUMBER         BIT (16),
      02 IPC.PGM.LR                 BIT (24),
      02 IPC.FORWARD.LINK           BIT (24);
```

IPC.RUN.UNIT.LIST is a linked serial list which includes all members of all Run Units. The entries in this list aren't in any particular order and are not grouped by Run Unit. The SDL portion of the MCP is responsible for the management and maintainence of all IPC.RUN.UNIT.LISTs. The first IPC.RUN.UNIT.LIST is addressed by a field in the MCP's stack. Both the S.MCP and the Micro MCP (M.MCP) access these structures for information.

IPC CALL OPERATION

The MICRO MCP receives all CALL communicates. Any named job is considered a candidate for a CALL by the Micro MCP. If the requested job is not currently a member of the correct Run Unit, then the CALL request will be transferred from the Micro MCP to the SDL portion of the MCP, to make the called program present.

To determine if the requested job is a member of the correct Run Unit, the Micro MCP searches the list of Run Units, beginning with the first, which is addressed by a field in the MCP stacks. Each program that is a member of any Run Unit will be found in the serially link list described by the IPC.RUN.UNIT.LIST structure.

If the program is present, the Micro MCP will first examine the program's RS.CANCELED boolean in its Run Structure Nucleus. If this boolean is true, then this copy of the program has been cancelled and a new copy must be initiated. CANCEL operations, like all program termination operations do not happen immediately. If a new copy must be initiated, the Micro MCP will call the S.MCP to initiate a new copy of the same program. S.MCP operations, upon receiving a CALL communicate, are described in later paragraphs.

If the RS.CANCELLED boolean is false, then the Micro MCP checks to determine whether the called job is active or passive, which will be indicated by the RS.RUN.UNIT.LINK field of the Run Structure Nucleus. If the job is already active, then some theoretically impossible error has occurred and the Micro MCP must call the S.MCP so that the Run Unit can be terminated. If

the called program is found to be passive, then the Micro MCP will next check to insure that the number of parameters to be passed, if any, agree. This will be indicated by the calling program's RS.IPC.DICT field being equal to the called program's IPC.PARAMETER.LIST.

If the number of parameters do not match, then the Micro MCP calls the S.MCP for termination of the run unit. If the number of parameters do agree, the Micro MCP next checks to insure that the length specified for each passed parameter is the same in the calling program and the called program. If any of the length descriptions are not equal, the Micro MCP will call the S.MCP for termination of the entire Run Unit.

If parameters are being passed, if the number of parameters is equal and if they all have equal length attributes, then in the calling program's Run structure Nucleus, the Micro MCP increments the RS.TEMPORARY.FREEZE field, to fix the program in memory and sets the RS.RUN.UNIT.LINK field to the caller's job number. In the Run Structure Nucleus of the called job, it sets the RS.CALLERS.LR field to the limit register of the calling program. It then hangs the calling program on its RS.IPC.EVENT field and sets the caller's RS.STATUS field to "waiting on the called program" and marks the called job "ready to run". It should be noted that it is not necessary to freeze the calling program in memory if parameters are not passed.

Considering the case where the called program is not a member of the Run Unit and the S.MCP is called upon to execute the requested program, whenever the S.MCP receives a CALL communicate and usercodes are involved, it will first search the list of Run Units, using all permutations of the usercoded name, to determine if the job exists in the Run Unit under a different name. if so, the new name and corresponding information will be entered into the Run Unit List and control will be returned to the Micro MCP. If Usercodes are not involved and if the name does not exist in the Run Unit List, then execution of the job must be attempted.

The S.MCP must then determine that the requested program is present on disk. If not present, the program which issued the CALL will be hung until the requested program is made present. If the requested program is present on disk, the S.MCP must then determine that there is enough memory to execute the requested program. If there is insufficient memory, The program which performed the CALL may have asked to be notified of this fact. If so, the called job will not be scheduled but the program which performed the CALL will be notified of the insufficient memory condition. If the program which performed the CALL did not request to be nofitied, the called program will be scheduled and the calling program will not be allowed to execute until the

called program does an EXIT PROGRAM communicate.

Actually, after the called program reaches BOJ, the S.MCP will hang the called program on his own RS.IPC.EVENT with RS.STATUS set to "Waiting to be called" and put the calling program back in the M.COMM.Q. This allows the Micro MCP to complete the CALL operation.

IPC CANCEL OPERATION

All aspects of the CANCEL and EXIT PROGRAM communicate opertors are handled by the Micro MCP. Upon receiving a CANCEL operator, the Micro MCP must first determine if the job exists in the Run Unit and whether it is active or passive. If it is not present or the program is present but its RS.CANCELED boolean is true, the request is ignored and the cancelling job is reinstated. If it is present and passive, the Micro MCP will then place the specified program in the EXTERMINATE.Q, set the RS.CANCELED boolean and return control to the job which issued the communicate. The EXTERMINATE.Q will cause termination of the job.

A request to CANCEL a job that is both a member of the Run Unit and active is a violation of the COBOL specifications and will result in termination of the entire Run Unit.

IPC EXIT PROGRAM OPERATION

If a called job issues an EXIT PROGRAM communicate operation, the Micro MCP will hang the issuing program on its RS.IPC.EVENT field, setting RS.STATUS to "Waiting to be called", decrement RS.TEMPORARY.FREEZE in the Run Structure Nucleus of the program that called the issuing program and mark the calling program ready to run. If a program that was not called issues the communicate, the communicate will be ignored and control will be immediately returned to that program.

IPC TERMINATION CONSIDERATIONS

If any program in a Run Unit performs a STOP RUN communicate operator, the entire Run Unit will be cancelled and all programs in the Run Unit will be discontinued. Similarly, if any program in the Run Unit terminates abnormally, the entire Run Unit will be discontinued. Programs within a Run Unit may only stop execution via the EXIT PROGRAM verb. Normal termination will occur when the program that initiated the Run Unit terminates.

Upon termination, for any reason, of any member of a particular Run Unit, the S.MCP will immediately delink all entries pertaining to the specified Run Unit from the Run Unit List. When the parent of a Run Unit goes to a normal EOJ, then all jobs attached to that run unit will be cancelled. If any job in a run unit is aborted, then the entire run unit will be aborted. If one program in a Run Unit does a CANCEL on another program in the same run unit, then the cancelled job must be delinked from the run unit and sent to EOJ.

IPC MICRO MCP/S.MCP COMMUNICATION

The transfer of information between the Micro MCP and the S.MCP is accomplished using an existing mechanism. This mechanism utilizes the Run Structure Nucleus field, RS.M.PROBLEM. All instances of such required communication are shown in the following table. The table shows the value that will be stored in the RS.MPROB.P2 field, a subfield of RS.M.PROBLEM, the condition that caused the communication and the action that will be taken by the S.MCP. Whenever such communication is necessary, the RS.MPROB.P1 field, also a subfield, will be set to a value of 9, which will indicate the family of problems related to IPC.

RS.MPROB.P2 =	Error Description	Required Action
0	Requested program not not in mix.	S.MCP should make program present.
1	Number of parameters do not match.	S.MCP will DS entire Run Unit.
2	Parameter specs. do not agree.	S.MCP will DS entire Run Unit.
3	Attempted recursive CALL	S.MCP will DS entire Run Unit.
4	Attempted CANCEL of predecessor.	S.MCP will DS entire Run Unit.
5	Invalid Communicate parameters.	S.MCP will DS entire Run Unit.
6	Found requested job and RS.CANCELED true	Terminate specified job and make new copy present.

IPC PROGRAM DUMPS

If any member of the Run Unit, regardless of whether the specified program is active or passive is DS-ed, DP-ed, or DM-ed, the entire Run Unit will be dumped and, if DS-ed or DP-ed, terminated.

IPC CANDIDATES FOR ROLL-OUT

After the Micro MCP processes an IPC communicate, it will not purposely mark the appropriate job TO.BE.ROLLED.OUT. If the S.MCP needs memory, it will follow the normal selection process in determining a candidate(s) for Roll-out. There will be no special consideration given to members of Run Units.

IPC TASK CONSIDERATIONS

Each called program will actually become a TASK associated with the originator of the Run Unit. By becoming a TASK as opposed to a normal job, several advantages will be realized.

1) TASKS are not subject to MIX limits and other scheduling constraints.
2) There will be corresponding entries in the SYSTEM/LOG.

IPC PROGRAM NAME SPECIFICATIONS

Information passed for the programs name in the CALL/CANCEL communicate will be subject to the same naming restrictions as a file with respect to DS or DP conditions, e.g., CALLing a program with a blank MFID will result in the termination of the entire Run Unit.

INDEX

ACCEPT 7-53
ACCESS DISK FILE HEADER (DFH) 7-43
ACCESS FILE INFORMATION BLOCK (FIB) 7-42
ACCESS FILE PARAMETER BLOCK (FPB) 7-41
ACCESS USERCODE FILE 7-62
ACCESS.GLOBALS 7-66
ACTIVE SCHEDULE 7-3
ANSII Tape Labels 3-26
Available Space Allocation, Index Files 3-74
Available Space Allocation, Indexed Sequential Files 3-72

Backup File "Early Start" Capability 3-50
BACKUP FILE BLOCKING FACTORS 3-49
BACKUP FILE CONTROL INFORMATION 3-51
BACKUP FILE LOGICAL RECORD FORMAT 3-52
BASE PACKS 3-40
Block Control Information - Indexed Sequential Files 3-62
Block Control Information - Relative Files 3-54

CA/RC CYCLES 3-8
CHANGE.ATTRIBUTES 7-66
CHANNEL TABLE 3-6
CLOSE 7-32
CLOSE (DM) 7-17
Cluster Files 3-59
COBOL PROGRAM ABNORMAL END 7-57
CODE FILES, PROGRAM PARANETER BLOCKS AND FILE PARAMETER BLOCKS 2-8
COMMUNICATE FORMAT 7-5
COMMUNICATES 7-3
COMPILE CARD INFORMATION 7-58
COMPILERS 1-3
COMPLEX WAIT (MICRO MCP) 7-64
Concurrent Update Operations 3-79
CONTENTION 5-1
CONTINUATION PACKS 3-40
CONVENTIONAL FILES 3-33
CREATE/RECREATE (DM) 7-46
CURRENT Maintenance 3-76
Current Record Pointer (CURRENT) 3-75

Data and Address Error Recovery - Disk Cartridges 3-84
Data and Address Error Recovery - 205 And 206 Drives 3-83
Data and Address Error Recovery - 207 Drives 3-83
Data and Address Error Recovery - 215 And 225 Drives 3-82
DATA AND FILE DICTIONARIES 2-12
DATA OVERLAY 7-42
DATA TRANSFERS 3-12

DC.INITIATE.IO 7-61
DCWRITE 7-62
DELETE (DM) 7-46
DEMAND MANAGEMENT 7-1
DESIGN PHILOSOPHY 8-1
Direct Files 3-54, 3-58
DISK DIRECTORY 3-37
DISK FILE HEADER 2-5, 3-37
Disk File Header Extensions 3-72
Disk File Header format changes 3-71
DISK FILE IDENTIFICATION 3-38
Disk File OPEN 7-28
DISK I/O CHAINING 3-14
Disk I/O Error Procedures 3-79
DISK I/O OVERLAPPED SEEKS 3-15
DISK IDENTIFICATION - PACK LABELS 3-24
DISK SPACE ALLOCATION 3-36
DISPLAY 7-53
DISTRIBUTION 5-1
DYNAMIC MEMORY BASE 7-59

EMULATOR TAPE (MICRO MCP) 7-55

FIB Dictionaries 3-67
FILE ACCESS AND IDENTIFICATION 3-37
File Identifiers 3-35
FILE INFORMATION BLOCKS 2-9
FILE NAMING CONVENTIONS 3-35
FILE STRUCTURES 3-33
FIND/MODIFY (DM) 7-45
FIRMWARE 1-3
FREE (DM) 7-50
FREEZE/THAW RUN STRUCTURE 7-58

GENERAL MEMORY MANAGEMENT CONCEPTS 4-1
GET SESSION NUMBER 7-61
GET.ATTRIBUTES 7-65
GISMO 1-3
GISMO - THE I/O DRIVER 3-4
GISMO/HARDWARE INTERFACE 3-7

HARD EVENTS 6-4
HIGH-PRIORITY INTERRUPT HANDLING 7-14

I/O ASSIGNMENT TABLE 3-18
I/O CHAINING 3-13
I/O CONTROL 1-5
I/O DESCRIPTORS 3-2
I/O INTERRUPTS 7-1
I/O SUB-SYSTEM 1-5
Index File Table Splitting 3-75
Index Files 3-58
Indexed Sequential Available Space Allocation 3-72
Indexed Sequential Buffer Descriptor (BD) 3-78

Indexed Sequential Buffer Management 3-77
Indexed Sequential Data File Available Space Allocation 3-72
Indexed Sequential Data File Structure 3-61
INDEXED SEQUENTIAL DELETE 7-69
Indexed Sequential Disk File Header Extension 3-72
Indexed Sequential File Global Information (GLOBALS) 3-69
Indexed Sequential Files 3-58
Indexed Sequential FPB 3-64
Indexed Sequential GLOBALS Field 3-69
Indexed Sequential I/O Descriptors 3-78
Indexed Sequential Index File Structure 3-62
Indexed Sequential Memory Structures 3-66
INDEXED SEQUENTIAL POSITION 7-67
INDEXED SEQUENTIAL READ 7-68
INDEXED SEQUENTIAL REWRITE 7-69
Indexed Sequential Structure Descriptor 3-71
Indexed Sequential Structure Descriptor (STR) 3-70
Indexed Sequential User Specific Information (USI) 3-68
INDEXED SEQUENTIAL WRITE 7-69
INDEXED/SEQUENTIAL OPEN 7-73
INITIALIZER I/O 7-51
INTER-PROCESS COMMUNICATION 8-1
INTER-PROGRAM COMMUNICATION 8-10
INTERPRETER 1-3
INTERPRETER MANAGEMENT, PARAMETER BLOCKS AND DICTIONARIES 2-5
INTERPRETER PARAMETER BLOCK 2-5
INTERPRETIVE PROCESSING 1-2
INTERRUPT RESOLUTION 1-5
INTERRUPT STACK 7-2
INTRODUCTION 1-1
IPC CALL OPERATION 8-17
IPC CANCEL OPERATION 8-19
IPC CANDIDATES FOR ROLL-OUT 8-21
IPC COMMUNICATE OPERATOR 8-16
IPC EXIT PROGRAM OPERATION 8-19
IPC IMPLEMENTATION OF SHARED DATA 8-11
IPC MICRO MCP/S.MCP COMMUNICATION 8-20
IPC PROGRAM DUMPS 8-21
IPC PROGRAM NAME SPECIFICATIONS 8-21
IPC Program Parameter Block Changes 8-14
IPC RUN STRUCTURE NUCLEUS CHANGES 8-12
IPC TASK CONSIDERATIONS 8-21
IPC TERMINATION CONSIDERATIONS 8-19
IPC Verb Operation 8-16
IPC.DICTIONARY 8-15

JOB SCHEDULING AND INITIALIZATION 7-2

LINAGE Clause 3-45
LINKED MEMORY 4-1
LOAD.DUMP MESSAGE 7-64
LOGICAL DISK FILES 3-36
Logical Page Function 3-45

M-MEMORY MANAGEMENT 5-1
MACHINE ARCHITECTURE 1-3
MCP OUTER LOOP 7-1
MEMORY DUMP TO DISK 7-59
MEMORY INITIALIZATION 4-10
MEMORY MANAGEMENT AND MEMORY LINKS 2-1
MEMORY REQUIREMENTS 4-15
MESSAGE BUFFERS 8-3
MESSAGE COUNT 7-65
MESSAGE DESCRIPTORS 8-3
MESSAGE.COUNT COMMUNICATE 8-7
MICRO/MCP 1-3
MINIMIZATION OF "CHECKERBOARDING" 4-4
MONITORING OF PERIPHERAL STATUS 3-18
MULTI-PACK FILE GENERAL RESTRICTIONS 3-42
MULTI-PACK FILE INFORMATION TABLE 3-41
MULTI-PACK FILES 3-39

N.SECOND 7-1
NDL/MACRO COMMUNICATES 7-62

OPEN 7-17
OPEN (DN) 7-16
OPERATING SYSTEM DYNAMIC REQUIREMENTS 4-21
OPERATING SYSTEM STATIC REQUIREMENTS 4-15

PACK INFORMATION TABLE 3-25
PACK LABEL 3-24
PE/NRZ EXCHANGES 3-31
PHYSICAL DISK FILES 3-36
POSITION (MICRO MCP (BACKUP FILES ONLY)) 7-38
PRINTER AND PUNCH BACKUP CAPABILITIES 3-48
PRINTER FILES 3-43
PRIORITY VICTIM SELECTION 4-7
PROCESS (PROGRAM) MANAGEMENT 6-1
PROCESSOR ALLOCATION 1-5
PROCESSOR I/O INSTRUCTIONS 3-8
PROG.IPC.MAX.SEND.PARAMS BIT(16) 8-15
PROG.IPC.PTR BIT(24) 8-15
PROG.IPC.SIZE BIT(16) 8-14
PROGRAM CALLER 7-63
PROGRAM COMMUNICATES 7-4
PROGRAM INITIALIZER 7-3
PROGRAM PARAMETER BLOCK 2-5
Program Parameter Block Changes 3-65
PROGRAM REINSTATE 7-4
PROGRAM-DEPENDENT DYNAMIC REQUIREMENTS 4-28
PROGRAM-DEPENDENT STATIC REQUIREMENTS 4-26
PROGRAMMATIC DETECTION OF MEMORY THRASHING 4-9

QUEUE ATTRIBUTES 8-4
QUEUE DESCRIPTORS 8-2
QUEUE DISK 8-2
QUEUE FILE FAMILIES 8-1

QUEUE FILE LOGICAL I/O OPERATIONS 8-5
QUEUE SYSTEM AND INTERFACES 8-1
QUICK QUEUE WRITE (REMOTE FILES) 7-62
QUICK QUEUE WRITE (STATION NUMBER) 7-62

RE-ENTRANT PROCESSING AND CODE SEGMENT DICTIONARIES 2-12
READ (MICRO MCP) 7-6
RECOVERY COMPLETE 7-65
Reference Address 3-4
RELATED DOCUMENTATION 1-1
Relative Disk File Headers (DFHs) 3-55
Relative File Buffer Management 3-56
Relative File Communicate Operators 3-56
Relative File Data Structure 3-54
Relative File Disk Initialization 3-55
Relative File Information Blocks (FIBs) 3-56
Relative File Parameter Blocks (FPBs) 3-55
Relative Files 3-54
RELATIVE I/O COMMUNICATE - DELETE 7-71
RELATIVE I/O COMMUNICATE - READ 7-72
RELATIVE I/O COMMUNICATE - REWRITE 7-71
RELATIVE I/O COMMUNICATE - START 7-70
RELATIVE I/O COMMUNICATE - WRITE 7-70
Remainder of the Disk I/O Error Procedure 3-84
ROUND-ROBIN VICTIM SELECTION 4-5
RS.CALLERS.LR BIT(24) 8-14
RS.CANCELED BIT(1) 8-14
RS.EXECUTE.TYPE BIT(4) 8-13
RS.IPC.DICT BIT(24) 8-12
RS.IPC.DICT.SIZE BIT(16) 8-13
RS.IPC.EVENT BIT (1) 8-14
RS.IPC.PARAMETER.LIST BIT(24) 8-13
RS.NAME CHARACTER(30) 8-13
RS.RUN.UNIT BIT(16) 8-12
RS.RUN.UNIT.LINK BIT(16) 8-12
RUN STRUCTURE 2-10
RUN STRUCTURE NUCLEUS 2-11
RUN UNIT DEFINITION 8-10

S-MACHINE 1-2
S-MEMORY MANAGEMENT AND MEMORY REQUIREMENTS 4-1
SDL MCP 1-4
SDL TRACE 7-55
SECOND CHANCE VICTIM SELECTION 4-6
SEEK (MICRO MCP) 7-12
SEGMENT DICTIONARIES AND SYSTEM DESCRIPTORS 2-3
SEQUENTIAL REWRITE (NMCP) 7-72
SERVICE REQUEST 3-9
SOFT EVENTS 6-1
SOFT I/O 1-5
SOFT MACHINE 1-2
SOFTWARE 1-3
SORT EOJ 7-58
SORT HANDLER 7-54

SORTER CONTROL 7-13
SORTER READ (MICRO MCP) 7-15
STATUS COUNTS 3-10
STATUS PROCEDURE 3-23
STORE (DM) 7-45
SWITCH.TAPE.DIRECTION 7-47
SYSTEM/REL.INIT 3-55

TAPE I/O CHAINING 3-16
Tape I/O Error Procedures 3-85
TAPE LABELLING, INITIALIZATION AND PURGING 3-26
TERMINATE (STOP RUN) 7-47
TERMINOLOGY AND DEFINITIONS 2-1
TEST.AND.WAIT I/O OPERATORS 3-23
The Error Correction Procedure 3-82
THE FENCE 4-3
THE I/O SUBSYSTEM 3-1
The Offset Procedure 3-80
The Strobe Procedure 3-81
TIME/DATE/DAY 7-50
TIMER INTERRUPT 7-1
TYPES OF MEMORY REQUESTS 4-2

UNIT MNEMONICS 3-22
USE/RETURN 7-54
USI 3-68

VICTIM SELECTION 4-4
VIRTUAL MEMORY 2-3

WAIT (SNOOZE) 7-52
WAITING SCHEDULE 7-3
WORKING SET DETERMINATION 4-6
WRITE (MICRO MCP) 7-9
WRITING TO THE TOP OF A QUEUE FILE 8-7

ZIP 7-52

9.0 Disk File Headers 3-71