Burroughs

Computer Management Systems (CMS) Master Control Program (MCP)

REFERENCE MANUAL

This Manual Replaces All Previous Issues of Form 2007555.

PRICED ITEM

Printed in U.S. America May 1982 Form 2007555

Burroughs believes that the software described in this document is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of use of this software material, including loss of profit, Indirect, special, or consequential damages. There are no warranties other than those expressly set forth in the PROGRAM PRODUCTS LICENSE AND SERVICE AGREEMENT.

The Customer should exercise care to assure that use of the software material will be in full compliance with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued from time to time to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to TIO Europe Documentation, Burroughs Machines Limited, Cumbernauld, Glasgow, Scotland G68 OBN

LIST OF EFFECTIVE PAGES

Page	Issue
iii	Original
iv	Blank
v thru xiii	Original
xiv	Blank
1-1 thru 1-3	Original
1-4	Blank
2-1 thru 2-15	Original
2-16	Blank
3-1 thru 3-14	Original
4-1 thru 4-10	Original
5-1 thru 5-34	Original
6-1 thru 6-7	Original
6-8	Blank
7-1 thru 7-43	Original
7-44	Blank
8-1 thru 8-15	Original
8-16	Blank
10-1 thru 10-26	Original
A-1 thru A-3	Original
A-4	Blank

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1
MCP OVERVIEW

INTRODUCTION

The Master Control Program (MCP) for the Burroughs Computer Management System (CMS), is the operating system responsible for managing the demands and resources of the system. The purpose of this manual is to provide an insight into the operation of the MCP on the host machine, to describe the various internal components of CMS and their relationship, and to show how they contribute to the overall management of the system.

OVERVIEW OF THE MCP

The primary function of the MCP is to provide effective and productive utilization of the resources of the CMS machines. External intervention is kept to the absolute minimum, and maximum throughput is achieved, by incorporating into the MCP the primary tasks of I/O control, file handling, multiprogramming, interrupt resolution and memory allocation, as well as providing an operator interface through intrinsic routines. For purposes of efficiency and flexibility, the entire MCP is written in machine micro code (machine language). The Computer Management System (CMS) MCP incorporates virtual memory techniques whereby the disk is treated as an overflow to internal memory. In fact, the MCP itself is mainly disk resident. It consists of many routines and functions, which interact with themselves and interpreters, to perform the tasks outlined in this manual. Because of the size of the MCP, only those segments of the MCP that are needed at a specific time are brought into memory. However, a portion of the MCP code is normally resident in memory. This includes the Device Dependent Routines (DDRs) which handle the keyboard printer combination, the disk, virtual memory routine, task scheduling and other frequently used routines.

OVERVIEW OF CMS

CMS is a disk based interpretive system which can be broken down into three hierarchical levels; the source level, the object (Secondary, or S) level, and the micro (machine language) level. Refer to figure 1-1. The source level is essentially the users interface with the system. This is accomplished through programs written in one of the high level languages provided: COBOL (Common Business Oriented Language), RPG (Report Program Generator), MPL II (Message Processing Language) and NDL (Network Definition Language). A program is compiled on the host machine, using the appropriate compiler, to produce an object or S program file. The compilers are written in MPL II, and are therefore MPL II S-program files themselves. All those S-files constitute the S or object level. An object file contains all the necessary information, arranged in a specified format, required for future execution by the appropriate interpreter. Interpreters are micro coded for optimal flexibility and efficiency. They interface with the operating system for some of their common functions. Therefore, duplication of common micro code between the various interpreters is eliminated, while specialized or supervisory and often accessed routines are gathered into the MCP. Together, interpreters and the MCP constitute the micro level which performs all the functions requested by the user. Because of the many interfaces between the MCP, interpreters, and S-program, object files must adhere to a standard format, and the communication between the interpreters and the MCP must conform to strict formats and procedures.

NOTE
COBOL and RPG share the same S-machine (Interpreter).

OVERVIEW OF THIS MANUAL

All input and output operations requested from the MCP are performed on files. The media supported by CMS depend on the host machine. However, certain peripherals are widely used, and therefore the

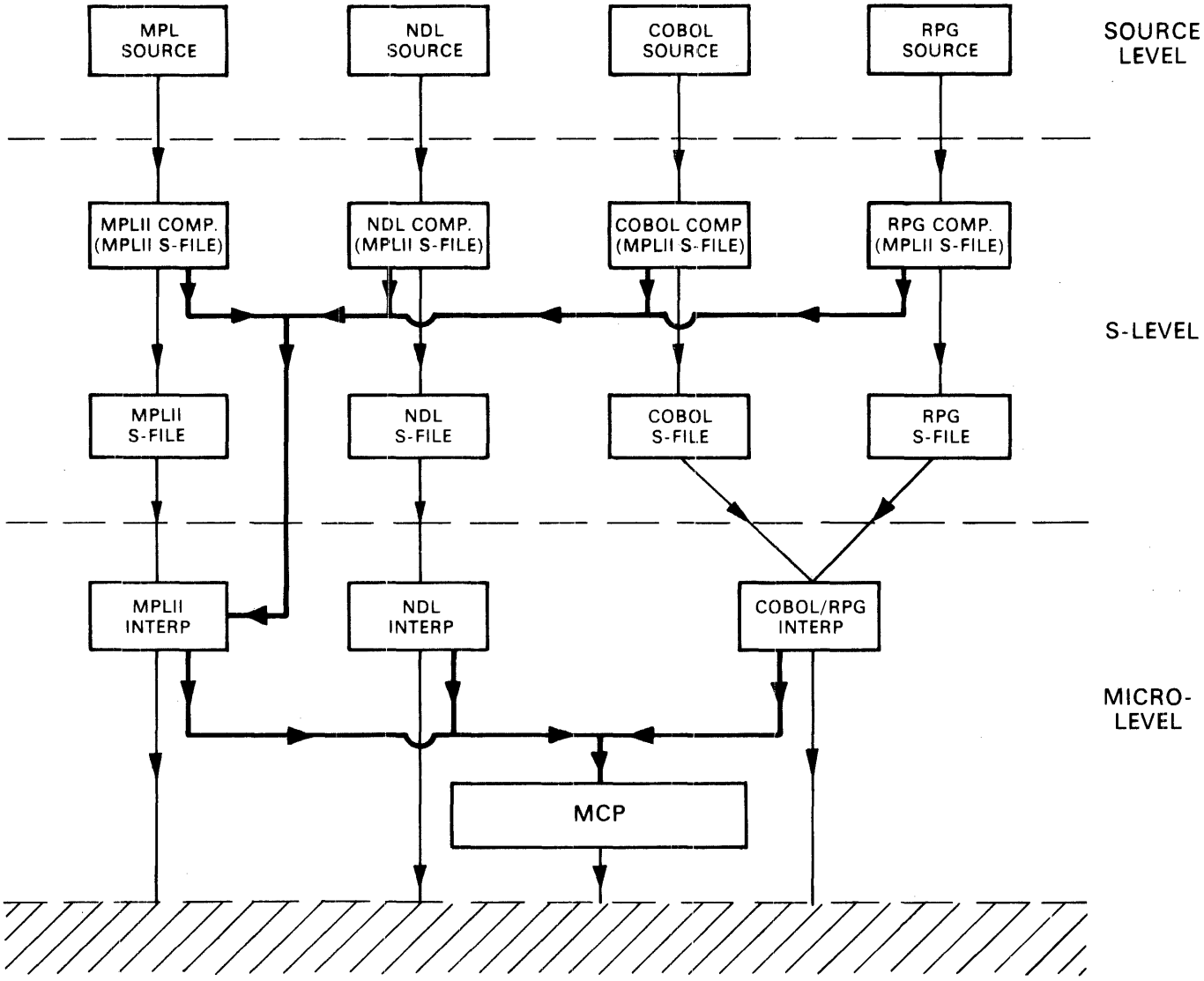


Figure 1-1. CMS Logical Levels

physical and logical characteristics of the files they support are described in Section 2. A logical file denotes the file as it appears to, and is accessed by, a user. A physical file refers to the physical format of the file on its storage media and is described in Section 3. The format of object files is detailed in Section 4. The communications between object files and the MCP are explained in Section 5. Virtual memory is discussed in Section 6. The various S-machines are discussed and a skeleton interpreter is presented in Section 7. Finally, those routines that are completely host machine dependent are expressed in the relevant host-dependent sections.

APPLICABLE CMS PUBLICATIONS

The following is a list of publications that are relevant to CMS and referenced in this manual by using the corresponding numbers in square brackets.

Reference Number	Publication Title	Form Number
[1]	CMS Software Operational Guide	2015228
[2]	CMS COBOL Reference Manual	2007266
[3]	CMS RPG Reference Manual	2007274
[4]	CMS MPL Reference Manual	2007563
[5]	B 90 Equipment Reference Manual	2017307
[6]	B 90 Operators Manual	2017240

SECTION 3
FILE ORGANIZATION AND ACCESS

INTRODUCTION

As described in Section 2, data, in units of logical records, is made visible from files to an S-program as a result of the execution of the appropriate communicate. The format and semantics of these communicates is the subject of a later section. Here, the structuring of data into files (File Organization), and their sequence of presentation to the S-program (File Access Method) is discussed.

THE FILE PARAMETER BLOCK (FPB)

Within any program file there is a data segment associated with each file to be used by the program. This data segment contains a description of the file as it appears from within a program. It is normally generated by the compiler after gathering the relevant information from the source program. Such information includes the file name, the volume-id (Multi-file tape or disk name), the file organization, and access mode. The format of an FPB is illustrated in table 3-1, and a sample is shown in figure 3-1.

In the paragraphs that follow, the various fields and their meanings are discussed in more detail.

```
00303030303030305046202020202020202020202030303100000000000A00000000
0084008400000002000832080000000000303030303030303030300000393939 00
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000
```

Figure 3-1. Sample FPB

IMPLEMENTATION LEVEL NUMBER

This field applies to disk files only and its value is used by the system to ensure that only disk files compatible with any particular release are handled by that release. This is done by comparing this value with the corresponding value in the files DFH.

VOLUME ID

This field is used by OPEN for disk and tape media to indicate which disk or multi-file unit contains, or is to contain, the file.

For disk, the disk id is used when opening any new or fully closed files. ASCII zeros in this field indicate the system disk.

For magnetic tape or cassettes, the multi-file id is used when opening or closing (beginning and ending labels) a new (output) magnetic tape, and when opening an (old) input tape or cassette. Zeros in this field denote a single file tape or cassette.

Table 3-1. FPB Format

Field Content	Location (Decimal)	Length (bytes)	Recording Mode
Implementation Level number	0	1	BINARY
Multi-file id/disk id	1-7	7	ASCII
File id	8-19	12	ASCII
Blank	20	1	ASCII
Reel/Cassette number	21-23	3	ASCII
File type (table 3-2)	24	1	BINARY
Highest record number within the file	25-27	3	BINARY
Device kind (table 3-3)	28	1	BINARY
Work area segment number	29	1	BINARY
Offset of work area in segment	30-31	2	BINARY
Record size in bytes	32-33	2	BINARY
Buffer size in bytes	34-35	2	BINARY
Maximum file size in bytes	36-38	3	BINARY
Number of buffers	39	1	BINARY
Flags (table 3-4)	40	1	BINARY
Adverb for close (table 3-5)	41	1	BINARY
Adverb for open (table 3-6)	42-43	2	BINARY
Cycle number	44-45	2	ASCII
Generation number (GN)	46-47	2	BINARY
Creation date	48-52	5	ASCII
Last access date	53-57	5	ASCII
Spare bytes in last stream file record	58-59	2	BINARY
Save factor	60-62	3	ASCII
* Data file disk id	63-69	7	ASCII
* Data file id	70-81	12	ASCII
* Blank	82	1	ASCII
* Spare	83	1	ASCII
* Rough table size in core **	84-85	2	BINARY
* Zero	86	1	BINARY
* Length of key in bytes	87	1	BINARY
* Offset of key within data record in bytes	88-89	2	BINARY
* Zeros	90-93	4	BINARY

* FPB extension for indexed files
** This field is not used for the CMS implementation on the B 90 family

FILE ID

The file id of a tape is restricted to seven characters while that of a disk may be up to 12 characters.

This field is required to open a fully closed file assigned to any device. For disk, this name is inserted into the disk File Directory Name List and corresponding DFH when closing a new file with lock. For tape, this field is used when opening (beginning and ending labels) a new (output) or an old (input) magnetic tape or cassette file.

REEL/CASSETTE NUMBER

CMS permits magnetic tape files to extend over many reels. For output, an end of reel label is written at the end of the reel (this end of reel label is distinguished from an end of file label by a single byte called the sentinel within the label), and a new reel is requested. Once supplied, the MCP will give it a label with the reel number incremented by 1. For input purposes, the MCP checks that the first reel mounted has a reel number (within its label) equal to this field within the FPB. This facilitates reading any reel of a multi-reel file. When the end of reel label is recognized, the operator is requested to supply the next reel and the MCP will check that the correct reel has been mounted.

FILE TYPE

The CMS MCP identifies and supports various types of files as exhibited in table 3-2, together with their associated code. With the exception of indexed files, all these files are sequentially organized. The file type is inserted into the corresponding DFH field when a new disk file is closed with lock.

The type of a file can only be changed when the file is fully closed. The various types are described individually in the next section.

Table 3-2. File Types

File Type	Code (Hex)
Normal Data	00
Source language	01-0E
Source Library	0F
S-code/program	10-12
Protected S-code	13 *
Interpreter (B 80/B 90)	14-16 *
Interpreter (B 900)	17 *
Interpreter (B 700/B 800)	18-1B *
Interpreter (B 1700/ B 1800/B 1900)	1C-1F *
Protected system files	20-2F *
SYSMEM	20 *
SYSLANGUAGE	21 *
SYSCONFIG	22 *
System related files	30-3F *
Dump files	30 *
Log files	31 *
System related	40-4F
Indexed (see notes below)	80
Keyfile	81
System related	A0-AF
Printer backup	A0
Filetypes marked * require special action when removing.	

HIGHEST RECORD NUMBER WITHIN FILE

This is set by OPEN from the corresponding field value in the DFH for disk and to zero for non disk files. It is maintained in the File Information Block (FIB) while a file is open. This value in the FIB is copied into the corresponding field in the DFH and the FPB when the file is being closed.

Related Note

The value #80 never appears on a DFH but is used in the FPB to indicate that an indexed file is being opened .

DEVICE TYPE

This field is required for all files. However, open sets the exact type of device assigned to a file in that field, which may entail changing the original value. This may take place when this field contains #02 for the printer, and in this case a code #0A is set by OPEN if an available ready line printer was allocated, or #07 if a console printer was allocated. A list of all the devices that can be handled by the CMS, together with their associated hex codes are shown in table 3-3.

Table 3-3. Device Types

Device Kind	Code (Hex)	Related Notes
PRINTER	02	1
SERIAL-PRINTER	06	
CONSOLE- PRINTER	07	
LINE-PRINTER	0A	
CARD READER (any)	11	2
CARD PUNCH (any)	12	2
CARD READER/PUNCH (any)	13	2
CARD READER (80 col)	15	3
CARD PUNCH (80 col)	16	3
CARD READER/PUNCH (80 col)	17	3
CARD READER (96 col)	19	4
CARD PUNCH (96 col)	1A	4
CARD READER/PUNCH (96 col)	1B	4
CONSOLE DEVICE	33	
CONSOLE SCREEN	3F	
ICMD	73	

WORK AREA SEGMENT NUMBER AND OFFSET

A record is made visible to the S-program in the work area supplied by that S-program for that file. This work area is specified to the I/O system as a data segment number, an offset within that segment and a length in bytes. Certain communicates (pertaining to Console and Stream I/O) require these parameters to be generated with every communicate. However, in general, these parameters are specified at OPEN time only.

The work area segment number and offset are taken to the File Information Block (FIB) at open time, and the work area length is the record length. The result of an I/O communicate is to move the contents of this work area to or from the appropriate part of the physical buffer holding the block (within the FIB).

#FF in the work area segment number field indicates to the MCP that no transfers are to take place to/from the work area.

(continued)

Table 3-3. Device Types

Device Kind	Code (Hex)	Related Notes
TAPE/CASSETTE (any)	81	5
TAPE/CASSETTE WRITE ENABLED	83	5
TAPE (any)	85	5
TAPE WRITE ENABLED	87	5
CASSETTE (any)	89	5
CASSETTE WRITE ENABLED	8B	5
ANY DISK (180 bytes/ sector)	C3	
BSMI (1MB)	C7	
207 FIXED PACK	CA	
CARTRIDGE	CB	
201I FIXED	CC	
211 FIXED	CD	
BSMII (3MB)	CE	
206 PACK	CF	

Related notes:

It should be noted that within the code, bit 6 is set to indicate output capability while bit 7 is set to indicate input capability.

It should also be noted that some of the above devices can only be implemented for some of the CMS host machines.

1. Output to this logical device is directed to a LINE PRINTER if one is available and ready. If not, it is diverted to the console printer, providing that one exists and is ready.
2. This specifies any card device, regardless of whether the cards have 80 or 96 columns.
3. This specifies 80 Column Card devices only.
4. This specifies 96 Column Card devices only.
5. This logical device is assumed to be a TAPE device if one is available; otherwise, it defaults to a CASSETTE device. 81-8B means NRZI or PE, A1-AB means PE, A1-AB means NRZI.

RECORD AND BUFFER SIZE

Buffer size must always be exactly divisible by the record size. Except for Console files, record size must be specified at OPEN time.

For a line printer device, a zero record size implies the physical transfer size. For a tape device, if the record or block size is zero, then the corresponding value is taken from the label.

For a new disk file, record and block sizes must be specified, and are then stored in the DFH. When opening an old file, if the block size in the FPB equals the block size in the DFH, or is zero, the value in the DFH is used. Otherwise, if both are multiples of 180 bytes, and the DFH (old) block size is a multiple of the FPB's (new) block size, the FPB's size is used. Otherwise a fatal error occurs.

For record sizes, if the FPB's record size is zero, the DFH's value is used; otherwise the FPB's record size is used.

Note that the chosen value of the block size is used to create the File Information Block (FIB) buffers.

MAXIMUM FILE SIZE

This field is only relevant to disk files and is otherwise ignored. It indicates the maximum number of records which can be written to that file.

A new disk file must provide the maximum file size in this field, and the figure must not exceed 2^{20}-16 records, or, in the case of single area files, 2^{16}-1 records. This figure is then copied to the DFH for future reference and is used in calculating the sizes of the areas to be allocated to the file. For old files, this field must contain #000000, otherwise a file is assumed to be a new one.

NUMBER OF BUFFERS

This is specified at open time except for Console and Stream I/O. The number of buffers may vary between 1 and 16 inclusive. Note that the buffers of input and I/O sequential access files are filled ahead of the program attempting to read records from them. This buffering ahead cannot, of course, be maintained if records are read by the program faster than they can be physically supplied by the device. Buffering ahead can also be disrupted if the START communicate is used in an unusual circumstance, such as sequentially updating a file containing unallocated disk areas.

Buffering ahead is not normally performed on random access files. However, if the program's accesses to the file are locally sequential, then, when this is detected, buffering ahead is performed on all available buffers. For sequential organizations, the onset of local sequential access may be detected by the observation that the new record requested by the program has a key one greater than that of the previous record. Accidental invocation of this feature by random use of a random access file generates little overhead, since files for random processing use few buffers. For Indexed files, the onset of sequential access can be assumed on the receipt of a read with the next adverb. Buffering ahead is discontinued when the program reverts to random access. Subsequent locally sequential access causes the loss of any previous buffering ahead.

On output and I/O files, buffers are queued for output (if necessary). For sequential access output only files, this occurs when the buffer is full. For random access and I/O files this occurs only when access is made to a record in another buffer.

FLAGS

The interpretation of the individual bits within this field, depending on whether they are set or not, is summarized in table 3-4.

Table 3-4. FPB Flags

Bit Set	Meaning	Related Notes
0	Duplicate keys allowed (Indexed)/ Forms required	1
1	Use FPB for last access and creation date update	2
2	No label	3
3	Conditional communicate	4
4	Single area required for new disk file	5
5	Generation number check or update	6
6	Non standard translate	
7	Reserved	

Related notes:

1. If this bit is set within the FPB of an Indexed file, then the MCP will not inhibit writing records with duplicate keys to that Indexed file. These records reside in both the data and key file constituents in the same order as they were written. Records with duplicate keys are also retrieved in that order.

   If this bit is set within the FPB of a printer, Console-Printer, or Line Printer file, the operator is requested via a SPO/ODT message at open time to select the output device by an 'AD' command.

2. For a tape file, if this bit is set then the value of the creation date field in the header label is updated from the value in the FPB at OPEN time, and in the trailer label at CLOSE time. For a disk file, if this bit is set, then the value of the creation date and last access fields in the DFH are updated from the corresponding values in the FPB at close time.

3. If this bit is reset, it implies a standard label. This is ignored except on magnetic tape, line printer, and output console. If set, it implies that the label of the file is omitted, and the operator has to select the output device by an 'AD' command.

4. Every communicate verb has two forms: a conditional and unconditional communicate. The semantics of each are discussed in Section 5.

   With the exception of OPEN and CLOSE, conditional and unconditional communicates can be differentiated by the setting of the least signnificant bit in their associated verb. This bit is used to indicate a conditional OPEN/CLOSE.

5. This field is available to instruct the MCP to place the file in a single area. The maximum file size, in this case, must be less than 2^{16} records. The compilers make use of this facility to ensure that all object files generated occupy a single allocated area.

6. For a full discussion of this bit, see the DFH related note number 7 in Section 2.

ADVERB FOR CLOSE

This is a byte field that follows the CLOSE verb in the CLOSE communicate. The setting of the various bits is illustrated in table 3-5.

Table 3-5. FPB Adverb for CLOSE

Bit	Value	Meaning
0		Not used
1	1	(see note below)
2,3,4	001	full close with release
	011	full close with lock
	101	full close with purge
	111	full close with remove
	others	half close
5	1	Crunch (cannot be extended later)
	0	Normal
6	1	merge overflow with index
7		Not used

Note: Bit 1 is used when tape file is half closed. If this bit is set and the multi-file id was not '0000000' at open time (not a one file reel), then the file is closed with no rewind. Otherwise, if this bit is reset, the file remains open, the reel is re-wound, and the operator is requested to supply the next reel.

ADVERB FOR OPEN

This is a two byte field that follows the OPEN verb in the OPEN communicate. The settings of the various bits are illustrated in table 3-6.

Table 3-6. FPB OPEN Adverb

Bits	Value	Meaning
0		Not used
1	1	Extend
	0	Normal
2,3,4	110	Normal, free access
	100	Normal, free access
	010	Lock access
	000	Lock
	001	Shared
5		Createnamedbackup flag
6		Output
7		Input
8,9		Backup option
10,11		Not used
12,13	00	Illegal
	01	Random
	10	Sequential
	11	Stream
14,15		Not used

'OTHERUSE' bits

'MYUSE' bits, at least one must be set (See table 3-7)

'ACCESSMODE' bits

Under free access, which is the normal situation, up to seven logical files can share (open and access) the same disk physical file, though only one user can have write access. Under lock access, only input users can open and access that file, but no output users can open it.

Under lock, no other users can open that disk file. (See 'shared disk files' in Section 2.)

'MYUSE' can assume the permissible values shown in table 3-7, depending on the device assigned to the file. In that table, I stands for input, O for output, and I/O stands for input output.

Table 3-7. Permissible 'MYUSE' Values

Device	Permissible 'MYUSE'
Console-Printer	I,O,I/O
Line-Printer	O
Card reader	I
Card punch	O
Card reader/ punch	I,O,I/O
SELF-SCAN® /CRT	I,O,I/O
Tape (write inhibit)	I
Tape (write enable)	I,O
Disk	I,O,I/O

SELF-SCAN is a registered trademark of Burroughs Corporation

CYCLE NUMBER

This field applies only to magnetic tapes or cassette files and is used to distinguish between different cycles of output tape files. It is used by OPEN and CLOSE (beginning and ending tape or cassette labels) for new output tape files, and is set from the corresponding label field when opening an old input tape or cassette file. Its value is never checked by the MCP and is for program use only.

GENERATION NUMBER

This field is described fully in Section 2 under the discussion of the DFH (related mode 7).

CREATION DATE

For a disk file, this field is set to the corresponding value in the DFH whenever opening an old or half closed disk file. It is set to zero when a new disk file is opened. It is copied back to the DFH if, and only if, the appropriate bit in the flags is set at CLOSE time. It is also copied into the DFH when a temporary disk file is closed with lock.

For a magnetic tape or cassette file, it is used by OPEN and CLOSE output of the file (beginning and end labels). The value however is moved from the label to this field when an old magnetic tape or cassette file is opened for input. In this case it can be accessed programmatically.

LAST ACCESS DATE

This follows the same rules as those indicated for the 'CREATION DATE', except that this field applies only to disk files (no corresponding field in the label of tape files).

SPARE BYTES IN THE LAST STREAM RECORD

With stream I/O, fields (not records) are accessed sequentially. The End of File (EOF) indication normally corresponds to the end of the last record within a file. With stream I/O, this is not sufficient, and the number of spare bytes in the last record is required to compute the correct EOF pointer.

SAVE FACTOR

This field is only meaningful for files assigned to a tape or cassette and is documentational for all other devices. It may assume a value between 0 and 999. It is added to the system date to give the purge date field in a tape label. When a tape or cassette is mounted, its label is automatically read by the MCP (see Section 2, 'INTRODUCTION', automatic volume recognition).

If the purge date in the tape label is less than the current system date, then the MCP internally marks that tape to be scratch and available for output. Note that no scratch label is written to the tape.

DATA FILE DISK ID

This field is used by the OPEN of an Indexed file which contains a new data file, and identifies the disk which contains the new data file. If an Indexed file with an old or half closed data file is opened, then this field is set from the corresponding key file parameter block (KFPB). Refer to table 3-8.

DATA FILE ID

When an Indexed file with a temporary data file is closed with lock, this name is inserted into the File Directory Name List and the corresponding DFH. If an Indexed file with an old or half closed

data file is opened, then this field is set from the corresponding Key File Parameter Block. Refer to table 3-8.

KEY LENGTH AND OFFSET

These fields are used to inform the MCP of the location and length of the field within the data record which is to be used as a key when opening an Indexed file with a new data file. When an Indexed file with an old or half closed data file is opened, this field is set from the corresponding Key File Parameter Block (KFPB). (Refer to table 3-8.) Zeros in these fields imply a key file with null keys, as created by the SORT utility [1].

CORE INDEX (ROUGH TABLE SIZE IN CORE)

The core index field allows for CMS implementations on hardware which is unable to perform stream disk searches to create and bring to memory a 'very rough' table. This is used when accessing an indexed file to limit that section of the rough table that needs to be searched to indicate the portion of the key file which contains the index to the data record. For the B 90, its value (which is the default value) is zero.

FILE ORGANIZATIONS

The CMS MCP supports three different file organizations: Sequential, Indexed and Console. Each of these organizations is described separately.

SEQUENTIAL ORGANIZATION

All files on devices other than disk can be regarded as being recorded on a serial medium; that is, sequentially organized. This implies that the position of each record on the medium is a monotonic function of its creation time. Disks are distinguished by their random access capability which makes it possible to present the records of a file to the S-program in any order, as specified by the value of a relative key whose value corresponds to the ordinal position in the file; the first record having a relative key of 1, and so on.

Because of this random access capability, it is not necessary for records to be stored on the disk in an order that reflects their order of creation; that is, records can be randomly written to a disk file.

A sequentially organized file, on disk, consists of a Disk File Header (DFH) plus up to 16 data areas. The DFH has the format shown in table 2-7. Areas are assigned only when required for output records, while reading from an unassigned area would yield undetermined data values (see 'DISK ALLOCATION' in Section 2).

INDEXED ORGANIZATION

An Index file consists of two transparent sequentially organized files; a key file and a data file. A user references an Indexed file by specifying the name of its key file, and all references to data records in the data file are made via a subsidiary data structure called an index. The index, together with a Key File Parameter Block (KFPB), a rough table, and a possibly empty overflow region combine to form the keyfile.

Each data record is distinguished by the value contained in a key field within the record itself. The size and position of this key field is determined at file description time. The index structure maps the value of the key to the disk address of the corresponding record. For sequential access, records are presented in order of increasing key value. For random access, the required key is supplied by the S-program and the appropriate record retrieved. Any number of other key files, and therefore Index or-

ganizations, can share the same data file. They are created (except possibly the first) and then maintained using the SORT utility [1]. The data file itself can be accessed as a sequentially organized file by referencing it using its own name (either programmatically supplied or it defaults to the concatenation of the key file name with 'QQ'), or by referencing one of its key files while requesting a sequential organization. Note that updating an index file via one of its indexes may invalidate all its other key files. The MCP offers no protection against the use of invalid key files. However, generation numbers, present both in the DFH's of the key and data files, and the FPB, and which are described in Section 2 under DFH related note 7, offer protection against using invalid combinations.

The Key File

The key file appears in the directory in the same manner as any other file. However, it is distinguishable from other files by the value in the file type field in its header. When a key file is constructed, the system selects appropriate area sizes according to the algorithm specified in 'DISK ALLOCATION' Section 2. The internal structure of the key file consists of a logically contiguous block of 180 bytes records.

The key file cannot have areas on a second disk due to rough table construction restrictions.

The Null Key File

If the field area of the record which is to be used as a record key is contiguous, then a key file is created in order to create a normal Indexed pair. If the record key is to be constructed from various non contiguous sub-areas, then a 'tag sort' is invoked on the concatenation of these fields. This results in a 'tag' file which consists of a list of relative record numbers in an order corresponding to the ascending key values in the data records. This tag file is given a Key File Parameter Block, and is used the same way as any other key file except that with null keys, a tag file can only be accessed sequentially.

The Key File Structure

As mentioned previously, a key file consists of four regions; namely the Key File Parameter Block, the index, the rough table and the overflow region. Each of these is described separately below.

THE KEY FILE PARAMETER BLOCK (KFPB)

This area occupies the first sector of the first area of the key file. The format and content of this sector are illustrated in table 3-8.

Table 3-8. KFPB Format

Content	Location (Decimal)	Length (Bytes)	Recording mode
Implementation level number *	0	1	BINARY
Spare	1-2	2	–
Disk-id of data file *	3-9	7	ASCII
File-id of data file *	10-21	12	ASCII
Blank	22	1	ASCII
Space for implementation defined link to data file	23-27	5	–
KFPB flags – true if bit set	28	1	BINARY
Bit 0: B 90 created rough table	–	–	–
Bit 1: reserved	–	–	–
Bit 2: reserved	–	–	–

(continued)

Table 3-8. KFPB Format

Content	Location (Decimal)	Length (Bytes)	Recording mode
Bit 5: Data file is a dual pack file	–	–	–
Bit 7: Duplicates allowed	–	–	–
Relative record number of start of rough table	29-31	3	BINARY
Length of rough table in sectors	32-33	2	BINARY
Spare	34	1	–
Relative record number of start of overflow region	35-37	3	BINARY
Size of overflow region in sectors	38-40	3	BINARY
Relative record number of start of index region	41-43	3	BINARY
Size of index region in sectors	44-46	3	BINARY
Spare	47	1	–
Size of key part in bytes *	48-49	2	BINARY
Offset of keypart from base of data record * in bytes	50-51	2	BINARY
Zero	52-55	4	BINARY
Spare	56-179	124	–
* See the corresponding definitions from the FPB earlier in this sector.			

THE INDEX

This region occupies one entry for each data record that existed at the time the index was created (the entries are sorted in order of increasing key value). Entries can be 8, 16, 24 or 32 bytes long, provided that the size of the user's key is less than 6, 14, 22 or 29 bytes respectively. The entries are packed 22, 11, 7 or 5 to the sector, respectively, with the remainder of the sector filled with trailing zeros.

The format of an entry in the index or overflow region is shown in table 3-9.

Table 3-9. Index Overflow Entry Format

Bytes	Content
0 to (entry size-4)	Record key value, left justified, binary zero filled
(entry size-3) to (entry size-1)	Relative record number within data file of keyed record.

Deletion of a data record is indicated by setting the key value field of its key file entry to zero.

The index starts on a sector boundary, and any residual space in the last sector of the index is filled with trailing zeros. In the event of the last record corresponding to an entry being deleted, the record key field in the entry is filled with binary zeros. The record number field is left unchanged so that the sequence of the key value is not disturbed. The last entry in each area of the index is a dummy with the key value set to all ones and the logical record field equal to 0. The dummy entry is put in at open time when creating a new file. SORT does not lose it.

THE ROUGH TABLE

A rough table provides a coarse index into the index region to enable the scanning of the index to be started at the most appropriate point. This minimizes disk access when searching the index. The rough table is built by OPEN when an Indexed organization file is opened and the user count changes from 0 to 1. It associates a key value with some suitably sized portion of disk within which the index entry corresponding to the key value will be found, if an such entry exists. The format of the rough table varies, depending on the host machine. However, certain rules must be adhered to. Each entry in the rough table must be formatted as shown in table 3-10 and must be the same size as the entries in the index or overflow. There is one rough table entry for each group of 32 sectors in the index region. Each entry contains the highest key value to be found in this group of sectors left justified, binary zero filled.

The entry also contains the lowest sector address in this group of sectors. The rough table is not updated if the record corresponding to the highest key in a group of sectors is deleted. The rough table is terminated by a dummy entry containing #FF in every byte of the key field and zero in the disk address field. The number of rough table entries is always less than or equal to (index size in sectors)/ 32 + 33.

Table 3-10. Rough Table Entry Format

O– (entry size 4)	highest key value is this group of index vectors, left justified, zero filled
(entry size-3) to (entry size-1)	lowest sector address in the group of index sectors

THE OVERFLOW REGION

When a new record is added to the file, an entry for it is placed in an overflow region situated at the end of the key file. Records that were added and then subsequently deleted may or may not have entries in the overflow. The format of entries in the overflow region is identical to that in the main index. The EOF pointer in the key file DFH points to the end of the last allocated area of the file. The entries in the overflow region are held in sorted order; that is, when a new entry is added, earlier entries with higher keys are moved up one slot to accommodate it. This process may result in the elimination of deleted entries. When an overflow entry is deleted, this is indicated in the same manner as for deleted index entry. The last entry in each area of the overflow region is a dummy, with all ones in the key value field and zero in the logical record number.

When the indexed file is closed, the contents of the overflow region, if any, are merged with the index; if MERGE in the FPB is set to true, this creates a new key file.

CONSOLE ORGANIZATION

This third organization should be strictly regarded as falling under sequential organizations. However, to simplify documentation, and to remove any ambiguities when the syntax and semantics of some specialized communicates are described, it is more convenient to think of the console as a separate organization.

This organization consists of the keyboard, which can be logically linked to a special character oriented output device, such as self scan or a console serial printer. It can be opened with 'MYUSE' input (keyboard to memory), Output (memory to console-printer) or I/O (keyboard to memory with echo on the printer).

A basic difference between console and sequential organization lies in the fact that console files are not record bound, but field bound; that is, input and/or output can be performed on parts of a record.

FILE ACCESS MODES

The access mode to a file denotes the order in which records are made visible to an S-program from a file.

For sequential organizations, the CMS supports three access modes; sequential, random and stream.

The records of a sequential access file are visible to the program in the same order (except as modified by start) as they appear in the medium, and a record counter or current record pointer in the File Information Block (FIB) associated is updated during each unconditional or successfully completed conditional READ or WRITE. The records of a random access file are seen in a totally program controlled order and the relative record number must be specified at the time of each class A communicate (see Section 5).

With stream access, each GET or PUT communicate accesses a number of bytes in the same order as they appear in the medium. The number of characters visible on each access is controlled by the program, but the order of their presentation is not.

For Indexed organizations, the CMS supports three access modes; sequential, random, and dynamic.

Sequential access is performed as previously described, except that records are retrieved in the same order as their keys appear in the index part of the key file.

Random access is performed as previously described, except that the key value must be supplied instead of the relative record number at the time of each class A communicate (see Section 5).

Dynamic access implies that the file can be accessed randomly, but an adverb 'NEXT' can be used to qualify the verbs READ and WRITE in order to effect sequential access.

For Console organizations, access is sequential only.

SECTION 4
FILE TYPES

INTRODUCTION

In this section, the various file types that can be supported by the CMS MCP (as illustrated in table 3-2) are discussed in the order of their corresponding codes. In addition, the format of LOAD/DUMP (or library) tape files is described at the end of this section.

DATA FILES

These are ordinary files with no special restriction on their size, record length, blocking factor, or content.

Disk space for these files is allocated in up to 16 separate areas, each capable of holding 2^{16}-1 records. When a new file is created, none of these areas is allocated; areas up to the maximum permissable are allocated when required. The size, as declared in the FPB at creation time (first OPEN), is rounded up to make it an integral multiple of a block. The extra disk space, however, is never accessible to any user of the file. Note that a file size cannot exceed 2^{20}-16 records, or, in the case of a single area file 2^{16}-1.

SOURCE FILES

These are ordinary data files with only two conditions; they must contain ASCII characters and their record size must be 80 or 90 bytes.

SOURCE LIBRARY FILES

(To be specified)

OBJECT FILES

A source language compiler is a special purpose program which accepts source statements in that language, and translates them into object code which is stored on disk, either temporarily or permanently, as an entity called a program (or object, or Secondary, or S) file. This object file forms the input to the loader whose function is to arrange the information contained in the object file in memory according to the format (run structure) suitable for the interpreter which is to execute the code.

The program file consists of 180 byte records blocked 1. The content of these is organized into items. An item is an entity such as code or data segment, or a segment table, and can be of variable length, but it always starts on a record boundary.

An object program must have a Program Parameter Block (PPB), a Code Segment Table (CST), and at least one code segment. Any other items which are not required, such as a Data Segment Table (DST) and data segments, can be omitted and the appropriate pointers in the PPB set to zero. The items within an object file must appear in the following order:

- Program Parameter Block (PPB)
- Items directly indexed by the PPB other than CST and DST, such as Code Control Block (CCB), preset area and the Internal File Name Block.
- Code Segment Table (CST)
- Data Segment Table (DST)
- Code segments in order of increasing segment number

- Data segments in order of increasing segment number

The above items are described separately in the following paragraphs.

PROGRAM PARAMETER BLOCK (PPB)

The first record, relative record number one, of all code files generated by a compiler is the PPB. It contains information relevant to the program and its required Environment. The format of the PPB is illustrated in table 4-1.

Figure 4-1 illustrates a sample PPB.

The functional description of these fields follows.

Table 4-1. The PPB Format

Content	Location (Decimal)	Length (Bytes)	Recording Mode
Implementation level number	0	1	BINARY
Program name	1-2	12	ASCII
Hardware system	13-14	2	BINARY (see below)
MCP Version	15-20	6	ASCII
Spare	21-24	4	
Interpreter disk id	25-31	7	ASCII
Interpreter name	32-43	12	ASCII
Compiler name	44-55	12	ASCII
Compilation date (YYMMDD)	56-61	6	ASCII
Priority class (see table 4-2)	62-63	2	BINARY
Data segment for initiating message	64	1	BINARY
S-program start address	65-67	3	BINARY
Code Segment Table (CST) Length	68-69	2	BINARY
Code Segment Table (CST) Location	70-71	2	BINARY
Data Segment Table (DST) Length	72-73	2	BINARY
Data Segment Table (DST) Location	74-75	2	BINARY
Task Control Block (TCB) Preset Area Length	76-77	2	BINARY
Task Control Block (TCB) Preset Area Address	78-79	2	BINARY
Partial Stack Length	80-81	2	BINARY
Code Control Block (CCB) Preset Area Length	82-83	2	BINARY
Code Control Block (CCB) Preset Area Address	84-85	2	BINARY
Task Control Block (TCB) Preset Extension Length	86-87	2	BINARY
Internal File Name Block Length	88-89	2	BINARY
Internal File Name Block Address	90-91	2	BINARY
Task Control Block (TCB) Area	92	variable	

Hardware system (first byte)

01-10 .B 80/B 90

11-20.B 776
21-39.B 800
31-40.B 900
41-50.B 1700
51-60,B 1800

(The Second byte is machine dependent.)

```
0042494C4F505420202020202042494C2E5245562E3920202030303030303030
42494C494E5445525020202044504C49492D4434202020203736303333310400
FF000000000600020001E000300580050C0190000000000000000C000700000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
```

Figure 4-1. A Sample PPB

Implementation Level Number

This one byte field serves the same function as the corresponding one-byte fields in the FPB and the DFH. For more information, refer to either description.

Program Name

This field contains a standard 12 character file name, as assigned by the user. For more details, refer to the compile utility [1]. It is normally the same as the program file name.

S-Language Name

This field is inserted by the compilers and is purely documentational.

Interpreter Name and Disk ID

These two fields are necessary for the loader to locate and identify the interpreter required to execute this object file.

Compiler Name

This field is inserted by the compiler and is purely documentational.

Compilation Date

This field is inserted by the compiler and equals the system's DATE on the day of the compilation. It can be accessed programmatically.

Priority Class

Under CMS, each task belongs to one of three classes: C, B, or A. Class C denotes the highest priority class of job and includes data communication programs which require urgent servicing by the processor for short bursts and spend the rest of their active life suspended. Class B denotes a medium priority class of jobs and includes utilities. Class A denotes the lowest priority class of job and includes all user programs.

The classification of the various priorities is made by the compilers. However, a utility (MODIFY) facilitates changing that classification [1].

Within each class, relative priorities to the tasks currently active are dynamically re-assigned by the MCP. Such dynamic re-allocation of priorities depends on the class. For A and C, the MCP raises the priority of those tasks that are I/O oriented (since I/O is done asynchronously, and therefore frees the processor). For Class B, the priority scheme is reverse historical (Last In First Out: LIFO). The appropriate number of tasks, belonging to each class, that can co-exist (multi-program) varies, depending on the machine hosting the CMS.

The setting and interpretation of the bits in the priority field is illustrated in table 4-2.

Table 4-2. Priority Field Format

Bits	Value	Meaning
0	1	Load only if mix suitable
1	1	Program may open filetypes 20-2F and use non-data files (see table 4-5).
2	1	Suppress BOJ/EOJ messages
3	1	Assign a utility mix number if possible
4	1	Load if no disk space (SYS-SUPERUTL only)
5	1	Class A }
6	1	Class B } mutually exclusive
7	1	Class C }
8,9	00	Non Data Communications program file
	01	Reserved
	10	NDL program file
	11	MCS program file
10	1	Program can access all file types except 20-2F
11	1	Program uses dual alphabet/reverse escapement Constructs
12-14	–	Reserved
15	1	Program uses Data Communications Constructs

Data Segment for Initiating Message

This field allows the program to specify a data segment into which part of the message that inititiated it as a task will be placed, starting at the first non-blank character following the part of the message relevant to the loader. If this PPB field contains #FF, the message will be discarded. If not, the contents of the field are interpreted as a segment number in the task's DST. If the segment is of type 2 (see DST), the MCP creates a segment of appropriate length. If the segment is of type 0, the message overwrites the appropriate number of bytes at the beginning of the segment and will, if necessary, be truncated at the segment limit. If the type is 1, the message is discarded.

S-Program Start Address

This three byte field is subdivided into two regions; a one byte code segment number and a two byte offset within that segment. These are required for the interpreter to identify the first S-operation in the program.

Code Segment Table Descriptor

The code segment table is a separate item within the object file, and contains pointers to, and descriptions of, the program's code segments. Each entry in this table is six bytes long, (see CODE AND DATA SEGMENT TABLES). Therefore, the length field for the CST contains six times the number of code segments, while the location is given in terms of the relative record number within the object file of the start of the item.

Data Segment Table Descriptor

These two fields have the identical format and interpretation as the corresponding Code Segment Table Descriptor.

TCB Preset Area

The Task Control Block (TCB) preset area specifies an area which will be copied by the loader to the appropriate location within the associated task's TCB, to be used as a possible pre-initialized write area by the interpreter.

The length of the TCB preset area is given in terms of bytes, while its location is given in terms of the relative number of bytes, from the start of the PPB to the start of this area. The content of this area is S-language dependent and is further described in Section 7. A detailed description of the TCB as implemented on the B 90 can be found in Section 8.

Partial Stack Length

The partial stack area specifies an additional work area for the interpreter which could be used as an S-program control stack. The format of this additional stack area is S-language dependent. This area is reserved by the loader in the TCB and cannot be preset to any particular value. Note that the size declared in the PPB should only reflect the additional area required by an S-program. Any greater length needed by the interpreter or the MCP should be supplied by the loader in addition to this specified amount.

CCB Preset Area

This area is provided on a per program basis (that is, one copy for all concurrent invocations of the source program) and is initialized to values set by the compiler. The loader, as part of its function in building the run time structure, copies this area into a specified area in the Code Control Block (CCB).

The length of this area is given in bytes while the address is given in terms of the relative record number within the object file of the start of this item. The content of the area is S-language dependent and is further described in Section 7. A detailed description of the CCB as implemented on the B 90 can be found in Section 8.

TCB Preset Extension Length

This area does not appear in the object file. However, it is added, initialized to zero, by the loader, to the bottom of the TCB preset area at run time. It can be used by the interpreter as an additional work area.

Internal File Name Block Descriptor

The Internal File Name Block (IFNB) is a separate item within the object file. Its descriptor in the PPB consists of two fields: a length field (in bytes) and a location field which expresses the position of the IFNB in terms of the relative record number within the program file of the start of the item.

TCB Area

This is the area which is referenced by the TCB preset area as previously explained.

Segment Tables

Both code and data are segmented. The compilers arrange the descriptors for code and data segments into two completely independent tables; the Code Segment Table (CST) and the Data Segment Table (DST). The CST and DST are held as two separate items within the object file. They contain a 6-byte entry for each segment descriptor, specifying the length of the segment, the physical record, and flags indicating its type. The format of the descriptor is given in table 4-3.

Table 4-3. CST or DST Entry Format

Byte	Name	Value	Function
0	Type	0	Ordinary code or data segment
		1	This data segment is a File Information Block (FIB)
		2	Reserved for construction at run time (bytes 2-5 are 0)
		3	uninitialized data segment
1	Flags	Bits 0-5	Reserved
		Bit 6	If set, segment is to be locked in memory
		Bit 7	If set, segment is Read/Write. Always reset for code
2,3	Location	–	Relative record number within the file of start of segment
4,5	Length	–	Length of segment, or if segment is FIB, then byte 5 contains the DST index of the associated FPB

For more details regarding the functions mentioned in table 4-3, see 'CODE AND DATA SEGMENTS' later in this section.

Figures 4-2 and 4-3 show examples of code and data segment tables that were referenced by the example in figure 4-1.

```
000000044007F0000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
00000000C000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000C000000000000000000000000000000000000000000000000000000000000000
000000000000000000000C000000000C000000000000000000
```

Figure 4-2. Sample CST Sector

```
0003000007D0010100000003010100000004000100050003F000100060003F0000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
```

Figure 4-3. Sample DST Sector

CODE AND DATA SEGMENTS

Code segments are held in the same way as read only data segments, but their descriptors are placed in the code segment table.

There are several different varieties of data segments which are distinguishable by a type byte and a flag byte in their segment descriptors:

A data segment which is initially empty and is to be used as work space by the program need not itself appear as an item within the program file.

Its presence (requirement) is indicated by its descriptor having a valid length field, a blank address field, and a type of 0 or 3.

The value with which type 0 segments are filled when made present is defined as 0. Type 3 segments are uninitialized and may contain anything when made present.

A type 1 data segment signifies that this segment is a FIB. This again does not appear as a separate item in the program file. The FIB's segment number is used to identify the file in all communicates that refer to the file. The S-program is unable to access the FIB directly, and any attempt to do so will cause a fatal error. When the file is opened, information from the FIB is used to construct the buffers and necessary control blocks in the FIB. When the file is closed, the buffers are dismantled and the FIB becomes nominal, containing only an indication of the file state, the segment number of the corresponding FPB, and, if 'half closed', a link to the peripheral or the DFH.

The segment number of the FPB corresponding to any given FIB is chosen by the compiler and is recorded in the FIB segment descriptor in the program file. This association between the FIB and the FPB segments cannot be broken or altered at run time.

More details regarding the FIB usage can be found in Section 5. Its physical layout for the B 90 can be found in Section 8.

Read Only data segments and Read/Write segments that contain preset values appear as items in the program file. Their descriptors contain both a length and the relative record number of the start of their item. The Read only flag is set to advise the system never to write to that segment, and that this segment need not be copied back to disk when the space it occupies in memory is reclaimed (that is, when it is overlayed), since the disk holds an exact copy of it. The lock flag is set to advise the system that when this segment has been accessed by the program, the system must not attempt contin-

ued execution of the program unless this segment is in memory. The setting of locked merely requests that the segment is always present, not that it should always occupy the same physical location in memory. This location should not concern the S-program, which accesses segments only by segment number and the offset within them.

INTERNAL FILE NAME BLOCK

This block is essentially a table of elements, one per file declaration in the program, each consisting of 30 bytes subdivided into three fields, as shown in table 4-4.

Table 4-4. IFNB Entry Format

Byte	Function
0	The DST index for the associated FIB
1	The DST index for the associated FPB
2-29	The associated file name

The IFNB is used by certain utilities to facilitate quick access to the FPB associated with a file within the program. The FIB index is there to permit run time access to the file as well.

Figure 4-4 shows a sample IFNB sector.

```
0103494E5055544649C2020202020202020202020202020202020202020202002 04
5046202020202020202020202020202020202020202020202020202020202029F001E
4500065B976100029F011D1600065B976DC1016DC201A100596DC1826110008D
4FA30004A401510007610004510004799151000548 5B97510002916100015 8A2
0300265100076D018401020130 81FFA4016D00840102010 0FFA4016D00840102
0100FFA401A402FFEAFFF2FFFAA5FFAB6D004193
```

Figure 4-4. Sample IFNB Sector

INTERPRETER FILES

CMS supports three interpreters: COBOL/RPG, MPLII, and NDL. Each interpreter resides on the disk in the same manner as any other disk file. However, an interpreter file lacks two items: the data segment table and data segments.

SYSMEM FILE

Every CMS disk file directory contains a file describing the entire physical disk, with a name of SYSMEM', and a special file type (@20@) which prevents access except to tasks of special priority setting, as indicated by the priority field in their program parameter block (see table 4-5). The DFH corresponding to SYSMEM is shown in figure 2 7.

VIRTUAL MEMORY FILES

The format of these files is implementation dependent and is discussed for the B 90 in Section 8.

INDEXED FILES

(See Section 3.)

KEY FILES

(See Section 3.)

Note that the MCP imposes certain protective procedures on its files. Depending on their types, only certain tasks as indicated by the priority setting in their PPB (table 4-1) can open and close with certain purge or lock files. These restrictions are summarized in table 4-5.

Table 4-5. Permissible File Usage

Fully Closed old Files	which can be opened by	with a "MYUSE" of	and then closed with purge or lock
Data	All programs	Any	Always
S code and source	All programs	input only if in use	Always
'SYSMEM'	Utilities only	Any	Never
ucode and VM files	Utilities only	input only if in use	not if in use
Key files	Utilities only	input only if in use	Always

LOAD/DUMP TAPE FORMAT

A load/dump tape/cassette or library tape/cassette is a multi file tape/cassette with n + 1 files, where n is the number of files dumped to the tape using the LD system function [1].

Each file except the first (refer to the following paragraph) within the library tape is called 'FLnnnnn', where nnnnn is a sequence number beginning at 00001 and increasing up to the value n. All files consist of 180 byte records blocked 1 as illustrated in figure 4-5.

The first file, called 'FILE000', consists of a header record indicating the version number of the LD utility that created it, the date of the dump, and the number of files on the tape. Further records in 'FILE000' are images of the Disk File Headers for all the files dumped in the order that they appear on the tape. These DFHs are placed there to serve as a directory which can be interrogated by utilities such as TAPED and TAPELR [1].

'FL00001' through 'FLnnnnn' are images of the files dumped, the first record of each being a copy of this file's DFH. Regardless of the disk file's record size or blocking factor, each disk file is dumped as 180 byte sector images.

FILE 000	Header record	DFH of FL00001	DFH of FL00002	_ _ _	DFH of FLnnnnn
FL00001	DFH of FL00001	sector images-of-FL00001			
FL00002	DFH of FL00002	sector images-of-FL00002			
FLnnnnn	DFH of FLnnnnn	sector images-of-FLnnnnn			

Figure 4-5. Library Tape Format

SECTION 5
CMS COMMUNICATES

INTRODUCTION

A communicate is a message of a particular format which is sent from an S-program to the MCP, requesting a function such as opening a file or writing a record. This message in contained within a block of contiguous bytes, referred to as the Communicate Parameter Area (CPA). Byte 0 of the CPA (the most significant) contains the verb which indicates to the MCP the function required. Byte 1 of the CPA contains the object whose nature depends on the verb. Most commonly it is the FIB index in the data segment table (for class A). The remaining area of the CPA contains the adverb whose length and content depend on the particular verb. Available communicate operations are divided into the following classes: those concerned with record type I/O (Class A), file assignment (Class B), field oriented I/O (Class C), data communications (Class D), and some miscellaneous communicates not directly related to I/O (Class E). Communicates belonging to each of these classes are discussed below. B 90 machine dependent communicates (Class F) are discussed in Section 8. Table 5-1 summarizes the verbs and their associated class and type.

On the completion of each communicate operation, 24 bits of result information are associated with the task, and can be accessed by the appropriate S-instruction which causes the interpreter to convert the information to some accessible form of S-data. The information is only available to be fetched until the next communicate is made by the task. Not all communicates supply useful result information but they all destroy the result information of the previous communicate. The format of the communicate response is given in table 5-2, while the applicability range is shown in table 5-3 and its associated notes.

When the system cannot honor a particular communicate, two general situations may arise. Either the request is illegal, which means that for classes A and B the operator is advised and the interpreter is informed that the run is to be abnormally terminated, or the request made, though legal, cannot be satisfied due to insufficient resources. In the latter case, if the communicate verb used is conditional (or, for class B 'OPEN' with the appropriate bit in the FPB flags set), the operator is not advised, the communicate has no effect, and a conditional failure bit is set in byte 0 of the communicate response, together with some reasons for the conditional failures in bytes 1 and 2.

FILE STATUS

Every file is in one of two fundamental states: open or closed. A file which is open can be accessed by a program using class A communicates. A file which is not open cannot be accessed.

Table 5-1. Communicate Verb Assignment

Value (Hex)	Assignment	Class/Type
01	OPEN	Class B/File Assignment
02	CLOSE	
10	DISPLAY/	Class C/Field Oriented I/O
	ZIP/	
1F	PAUSE	
20	ACCEPT	
30		Class D/Data Communications
3F		

(continued)

Table 5-1. Communicate Verb Assignment

Value (Hex)		Assignment	Class/Type
40		DATE/TIME	Class E/Miscellaneous
41		TERMINATE	
42		WAIT	
50		NONE	For Future Use
6F			
70			Class F/ Machine Dependent
7F			
80		TEST STATUS	Class A/Record Type I/O
82	Not Console	READ	
84		WRITE	
86		REWRITE	
88		DELETE	
8A		STREAM CONTROL	
8C		START	
8E		OVERWRITE	
90	Console	READ-WRITE	Note 1: Valid only for open files Note 2: Appropriate Verbs may be made Conditional by adding 1 to verb value
92		READ	
94		WRITE	
96	Stream I/O	GET	
98		PUT	
A2		READ	Class A returning 209020 on shared block locked
A4		WRITE	
A6		REWRITE	
A8		DELETE	
AE		OVERWRITE	

Table 5-2. Communicate Response Format

Communicate Type	Byte	Value (Hex)	Meaning
Common	0	80	Fatal error; for example, invalid communicate
		40	Response temporarily unavailable (conditional failure)
		20	Abnormal result – further information is contained in bytes 1 and 2
		10	No message after data-communication dequeue
Class A	1 (Status Key 1)	00	Operation successfully completed
		10	End of file encountered on input for sequential access file
		20	Invalid key – reason given in byte 2
		30	Permanent error on this file
		90	Reserved for expansion

(continued)

Table 5-2. Communicate Response Format

Communicate Type	Byte	Value (Hex)	Meaning
Class A	2 (Status Key 2)	00	No further information
		10	Sequence error on output to indexed file (if 1=30 read error on data file)
		20	Duplicate key on indexed file (if 1=30 write error on data file)
		30	No such record exists (attempting to read randomly beyond EOF) (if 1=30 read error on key file)
		40	Boundary violation (attempting to write beyond allocated area) (if 1=30 write error on key file)
		41	Excess characters input on a console READ-WRITE
		CK	Control key used as terminator in a successful console input
ZIP	1	00	Load request or successful non-load request
		10	Valid but unsuccessful non-load request
		80	Invalid non-load request
ZIP	2	10	Load request failure – program file not found
		20	Load request failure – interpreter file not found
		30	Load request failure – no memory
		40	Load request failure – no user disk
		50	Load request failure – mix full
		60	User count error
		70	Duplicate pack
		80	Invalid load request
		90	MCS already present
		A0	Disk error
		B0	Code file error
		C0	Illegal data comm load request
		F0	Null mix required
		F1	Dual alpha/reverse escapement not supported
		F2	Insufficient real store
		F3	Disk locked
OPEN Conditional Communicate	1 & 2	000	MCP event number (see note [1])

Table 5-3. Communicate Response Applicability Range

Status Key 1 / Status Key 2	00	10(2)	20	30	90
00	SS,SR IS,IR C	SS IS		SS,SR IS,IR C	
10(1)			IS,IR		
20	IS,IR		IS,IR		
30(3)			SR,IR IS		
40(1)	SR		SS IS,IR	C	
41				C	
KEY	C				

SS: May be generated by Sequential organization and sequential access
SR: May be generated by Sequential organization and random access
IS: May be generated by Indexed organization and sequential access
IR: May be generated by Indexed organization and random access
C: May be generated by Console organization
(1): May be generated after an output operation only
(2): May be generated after an input operation only
(3): May be generated after an input or delete operation only

Many attributes of a file are bound at open time and influence the semantics of all communicates up to and including the next close (refer to table 3-7).

The closed file state has two subdivisions which depend on the file's previous history of opening and closure. These states are called 'Closed' and 'Half Closed'. When 'Half Closed', a file declaration is associated with an actual file; when closed,it is not. Open is valid for closed or half closed files and changes them to open. 'MYUSE' value as specified in the FPB interacts with the device specified and incompatible choices will cause a fatal error during open. Similarly, any requirements for random access capability for a device other than disk cause open to fail. Close is only valid for open files and changes them to closed.

The allowable transitions between new files and old files, and between open, closed and half closed files is shown in figure 5-1.

The actions that take place apart from logical file open or close are indicated at each transformation.

FILE ASSIGNMENT COMMUNICATES (CLASS B)

Syntax

The communicate S-operator is associated with a CPA which has the following format:

Verb (byte 0) = 01 : OPEN
= 02 : CLOSE
Object (byte 1) = data segment table index of associated FIB.

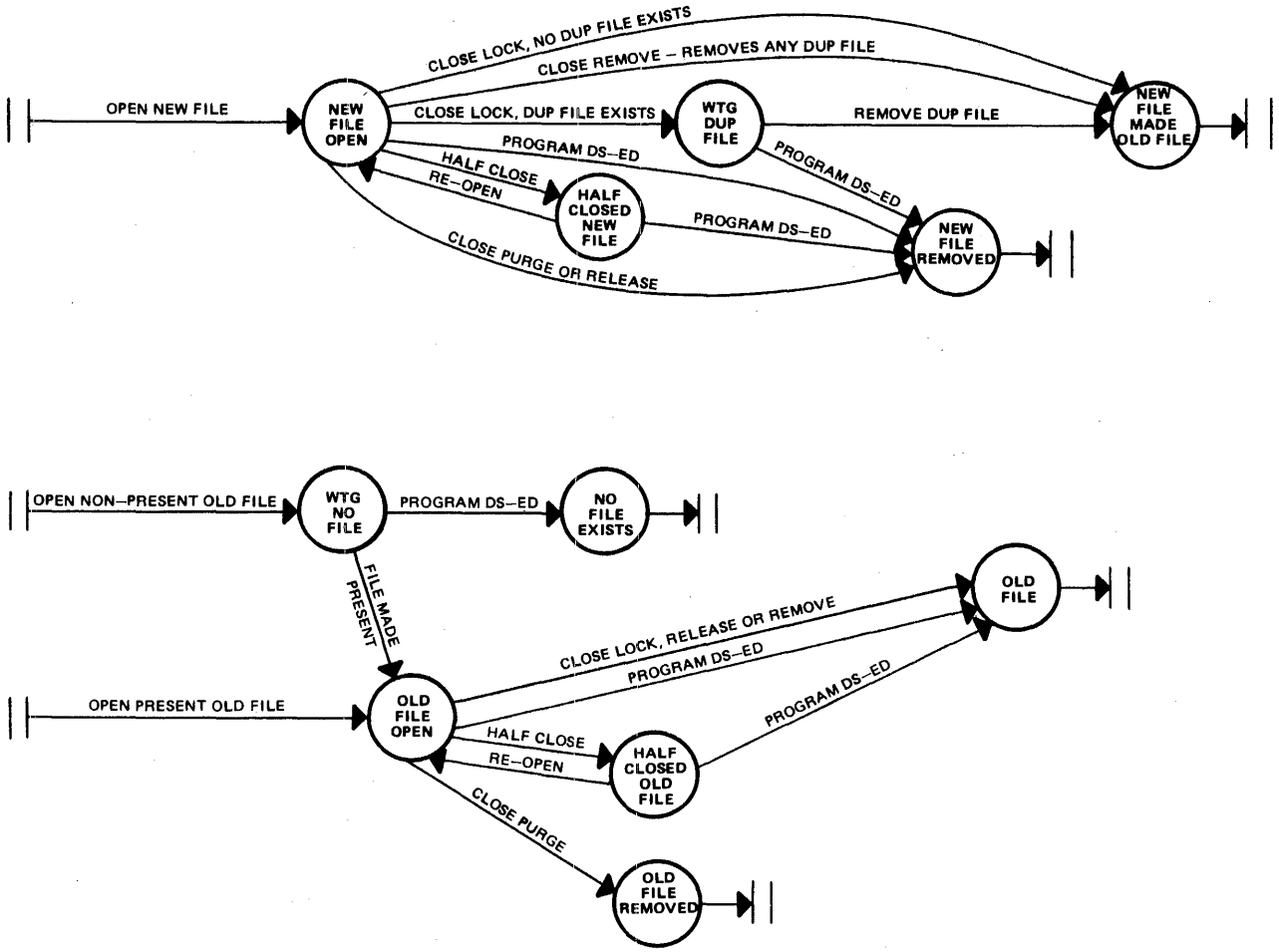


Figure 5-1. Open and Close States and Transformations

Adverb = parameters are contained in the OPEN and CLOSE adverb fields within the FPB (tables 3-6 and 3-5 respectively).

OPEN SEMANTICS

The semantics of the open verb depend on the file attributes as defined in the FPB. In particular, the values of 'MYUSE' and 'ACCESSMODE' (table 3-6) that do not correspond with the permitted capabilities of the device cause a fatal error during open. 'OTHERUSE' (table 3-6) is ignored unless the file is assigned to a disk. 'EXTEND' only applies to output assigned to disk or magnetic tapes.

The semantics of the open verb also depend on the current file state (closed or half closed). If the file is fully closed, then the configuration table (see Section 2, Introduction) is searched for a device of the correct name. Its kind must match that specified in the FPB. However, certain types can be satisfied by other types. For example, a request for a card punch can be satisfied by a punch or by a reader/punch. As a rule, the priority is given to the device with the fewest facilities or smallest data, in that order. The criterion for matching the name varies with the device kind and 'MYUSE'. All non disk files require a scratch marker (which is overwritten with the file id or multi file id for output tape files), while all disk and all input multi files require an entry matching the FPB file id.

When opening a fully closed file, inability to find a correct unit for any file, or the existence of a plurality of correct units for any disk or input non disk file, causes a message to be output to the operator and the suspension of the task until either the operator discontinues it, or the operator or another task rectifies the situation. A plurality of correct units for an output non disk file causes the system to make an arbitrary (though repeatable) choice. However, if conditional open is made (by the appropriate setting of the corresponding bit in the FPB flags), then if a file cannot be found, the task is not suspended, but a communicate response indicating 'resource temporarily unavailable' is returned to the calling task and the file state remains closed. The same action occurs if the open is unsuccessful due to a duplicate file condition. The two values are distinguished by the value in byte 1 of the communicate response. Note that for non-disk files, we do not distinguish between open failures due to the file not being mounted and failures due to the file being mounted but already assigned to a task. Both are treated as file not found. A permanent I/O error during open is fatal, even if the open was conditional.

If a fully closed file is legitimately declared to be unlabelled in its FPB, open waits at the appropriate point for a physical unit of the appropriate type to be assigned by an operator 'AD' SCL command. The operator cannot preempt the demand by open.

If a file is half closed, no configuration table search is needed. However, a test is made to ensure that the assignment recorded in the configuration table entry associated with the (vestigial) FIB is to the correct task and file declaration. This test protects against an operator unloading a half-closed file and subsequently mounting another file of the same name which is then opened via another file declaration. In addition, the name in the configuration table must be correct, unless a scratch marker is expected. Apart from this, any mismatch in name or assignment is fatal to the task.

When a device has been found or checked, open will construct a new FIB with suitable work areas including a tank of buffers with I/O descriptors, as specified in the FPB (and further described in Section 8), and a file state reflecting the success of the open.

A work area specification implying access beyond the segment size, or to an invalid segment, is fatal, as is a request for more than 16 buffers. A file can be opened with zero buffers, but any subsequent class A communicate is fatal. If the device requires labelling, open for output constructs and writes labels, but open for input copies label information to the FPB. Device dependent semantics are discussed below.

Non Magnetic Device Files

Card input cannot be half closed. Input and output to 80/96 column cards and output to line printer have default record and block sizes equal to the physical capability of the device. The default is invoked by specifying a zero in the appropriate FPB field. If the declared block size is greater than the physical capability of the device, a fatal error is returned.

If the FPB specifies an unlabelled line printer or console, only the data from WRITE statements appears on the media.

Magnetic Tape and Cassette Files

If the FPB multifile-id is '0000000', the declaration is treated as a single file tape. When opening such a file for input, the FPB file-id is matched with the configuration table. Otherwise, it is a multifile tape and the following differences apply:

1. Open for input on a closed file causes the configuration table to be searched for a unit loaded with that multifile tape. If the search is unsuccessful, a message is output to the operator who has the usual options. If the search for the multifile is successful, then the tape is physically

searched for the required file. If successful, the tape is positioned between the label and the first record of the target file. If unsuccessful, the tape is rewound and the operator informed.
2. Open for input on a half closed file causes the same actions, except that the configuration table entry is only checked and physical searching of the already associated multifile immediately commences (without rewind).
3. Open for output on a closed file is standard, but it is the FPB multifile-id which is copied into the configuration table.
4. Open for output on a half closed file assumes that the associated unit is positioned correctly. If the tape is not currently rewound, then the multifile id in the FPB must be equal to the multifile id in the configuration table, and an error is fatal to the task.

When opening any magnetic tape file, the reel number is regarded as part of the relevant identifier for matching, checking and copying.

If the EXTEND flag is set during an open output, an open input is performed instead (with semantics appropriate to the current FPB content). Then the tape is wound forward, stopping just before the terminating tape mark of the file, and the filestate set for writing, allowing the file to be extended. Any data previously written to a subsequent part of the tape is jeopardized. The EXTEND flag is ignored during an open input.

When opening an input file, the record size in the FPB is used, unless it is zero when the record size in the label is used. If this is also zero, a fatal error occurs. The block size in the FPB is used, unless it is zero when the block size in the label is used. If this too is zero, the chosen record size is also used as a block size.

A fatal error occurs if:

- either the FPB record size or FPB block size is zero and the file is declared to be unlabelled and/or non external.
- or the chosen blocksize is not an exact multiple of the chosen record size.

If the FPB specifies an input unlabelled tape, then 'AD' gives the program the exact contents of the tape, whether or not that includes recognizable labels. Each tape mark causes EOF. Reading can continue after half closed no rewind and open input. Reading through EOT causes program failure.

If the FPB specifies an output unlabelled tape, 'AD' can only indicate a validly labelled scratch tape, so output tapes cannot be assigned before the device has been made ready and the tape checked to see if it had such a label and also write capability. Only the data from write statements appears on the tape; each close writes one tape mark. Writing can continue after half close no rewind and open output. Writing through EOF causes program failure.

Disk Files

Some of the implications of the disk open verb are illustrated in figure 5-1. When opening a fully closed file, a non-zero FPB maximum file size is regarded as a request to create a new file. A zero value means find an old file. The value is irrelevant to opening a half closed file.

The disk id is used during every opening from fully closed (including a new file) to locate the desired disk unit. If the disk id is seven ASCII zeros, the disk containing the executing MCP is assumed to be specified.

The file id need not be specified for opening a temporary file, but must be placed in the FPB before any close with lock, since it will be used as the argument of a directory search.

An Indexed organization open with an output capability automatically provides the data file with lock access (no output users) and the key file (if any) with lock (no other users) unless SHARED is specified.

A task opening a disk file can specify whether it will co-exist with other attempts to open the file. This is done by indicating (in the OTHERUSE field in the FPB Adverb) the value of MYUSE which other attempts can use. This is specified in 'ADVERB FOR OPEN' in Section 3. If conflict arises, the task making the attempt is suspended until the file is closed and another attempt can be made. This mechanism only applies to disk files and is ignored for other devices.

When a file is half closed, the OTHERUSE access of that file is freed, although the user count is not decremented (that is, a half closed user looks like an open input user).

A task opening 'SYSMEM' with non-free OTHERUSE applies constraints to all other users of the same disk.

If a conditional open (through the appropriate setting of the corresponding bit in the FPB flags) cannot be successfully completed, the task is not suspended, but is restarted with a communicate response reflecting 'resource temporarily unavailable'. The reasons for this conditional failure are explained in bytes 1 and 2 of the communicate response. The possibilities are:

1. No file on the named pack with the specified file-id.
2. User count problem; for example, too many users, other user has lock, already one output user.
3. Illegal mismatch of FPB and DFH filetypes.
4. No space left in directory for another new file.
5. Program cannot open this filetype this way.
6. No pack with the specified disk id.

The duplicate file message shown in table 5-2 is not relevant to disk, since open must specify the disk id.

Sequentially organized disk files which are opened for output only with sequential access may be opened in an EXTEND mode. This is signified by a bit in the FPB, which is ignored under all other conditions. In this mode, the file is positioned as if the last action on the file had been to write the last existing record. This means that subsequent writes add new records to the end of the file.

Disk space for the file is allocated in up to 16 separate areas. Each area can hold up to 2^{16}-1 records. When a new file is created, none of these areas is allocated: areas up to the maximum permissible are allocated when required. The total size of the file in records must be specified in the appropriate FPB field when the file is first opened and created as new file. The size is rounded up if necessary to be an integral number of blocks, but the extra disk space is never accessible to any user of the file. A request for a maximum file size in excess of 2^{20}-16 causes a fatal error.

Record size and buffer size (which must be an integer multiple of record size) must be specified when creating a new file, and these values are stored into the file header. When opening an old file, if the block size in the FPB is equal to the block size in the file header or equal to zero, the file header is used. Otherwise, if both values are exact multiples of 180 and the header block size is an integer multiple of the FPB block size, then the FPB value is used, otherwise open fails fatally.

If the record size in the FPB is equal to zero, the file header value is used. If not, the FPB value is used. If the chosen block size is not an exact multiple of the chosen record size, open fails fatally. It is not permissible to open an old indexed file with a record size different to that with which it was created.

The chosen value of block size is used to create buffers in the FPB.

Opening a fully closed old disk file whose DFH filetype is 81 (keyfile) causes various actions, depending on the FPB filetype, as follows:

1. 80 (indexed): An indexed file is opened using both the keyfile and the file named in the KFPB (see table 3-8) which must have a DFH filetype of #00. An extended FPB (see table 3-1) must be used.
2. 81 (keyfile): the file is opened if the program is a utility; otherwise, open fails fatally.
3. Any other value: open transfers its consideration totally to the file named in the KFPB, after which that file's DFH filetype governs.

If the DFH filetype is not #81 it must equal the FPB filetype (unless the latter is #FF which is 'wild') and must also be compatible with the PPB 'SYSMEM' flag and the proposed MYUSE, as described in table 4-5.

Opening a fully closed new file ignores all filetypes, except that an FPB request for an indexed file causes the creation of a data file/key file pair using an extended FPB.

The filetype is totally irrelevant to opening a half closed file: the organization is stored in the vestigial FIB.

CLOSE SEMANTICS

Close can only be validly applied to a file which is open. Attempting to close a closed file causes a fatal error. There are two basic variants of close: full close and half close (see table 3-5).

Both variants dismantle the structure of FIB and buffers built by the matching open. The buffers are not dismantled until all queued output operations are completed. Files open output or I/O may require an additional output operation to amend a partially filled block to the file. A permanent I/O error inhibits further close processing except to return a fatal fetch message value.

The FIB is reduced to a vestigial form which contains only the filestate indicating half-closed or closed, the DST index of the FPB and, in the case of half-closed alone, a pointer to the device (and in the disk case, the file). A non-disk device is only released back to the system pool on a full close. When any close frees a serially-reusable resource (for example, a device, directory space, non-shared use of a file) the task suspended in OPEN for lack of that resource is automatically stimulated to try again.

Device dependent semantics are discussed below:

Non Magnetic Device Files

Closing a paper tape output file causes the punching of a trailer label, unless the file is declared to be unlabelled.

Closing an unlabelled paper tape input file causes any remaining media to be flushed.

Closing a card input or labelled paper tape input file before the end-of-file detection causes flushing of media until that detection is obtained.

Closing a card output file causes the punching of a trailer label or some form of end card.

Closing a printer output file causes the printing of a trailer label unless the file is declared to be unlabelled.

Closing a console output or I/O file causes the printing of a trailer label.

Half close of a card input file or a labelled paper tape input file is treated as close with release.

Close with purge is treated as close with release.

Close with remove is treated as close with release.

Close with lock implies that the device is marked as not ready and cannot be assigned via another open without being made ready by operator intervention. The state of the crunch flag is ignored. The state of the no rewind flag is ignored.

Magnetic Tape and Cassette Files

When a magnetic tape or cassette file is closed, the normal action is to rewind to the beginning of the tape. The action of rewinding is performed asynchronously, and when complete, the system treats the unit as it would a freshly loaded one; that is, reading the label and entering the name into the configuration table. This rewind action is inhibited if the no rewind flag is set and the multifile identifier close is requested. All the full-close options return the unit to the system. In addition, close with lock causes the unit to be marked not ready.

Close with purge causes a scratch label to be written at the beginning of the tape (even with unlabelled files, which provides a mechanism for labelling virgin tapes). Note that since the presence of write permit hardware will not have been checked at open time for an input file, it may not be possible for close with purge to write the scratch label. In this case, the system prints an operator message and treats the operation as close with release.

Close with remove is treated as close with lock.

For magnetic tapes the adverb for close offers the option of CLOSE REEL. This causes the closing of the current reel of the file and the subsequent opening of the next reel. The file id used for the search for the next reel is that held in the FPB. The reel number is one greater than the current FPB value. A CLOSE REEL on an unlabelled file causes program failure. If a CLOSE REEL is issued for a input file and on examination of the trailer label it proves that there is no next reel, then the condition is remembered and EOF is given on the next READ on that file.

When an output file is closed, the system writes a trailer label unless the file is declared unlabelled. In this case only a tape mark is written. When an input file is closed, the trailer label is checked only if end of file has been reached.

Close no rewind always leaves the tape positioned just after the current file (by moving the tape forward if necessary), ready to read the next file if one exists. If the file is declared as unlabelled, then no trailer label check is performed, but close no rewind still leaves the tape positioned at the end of file.

Disk Files

Some of the implications of the disk close verb are illustrated in figure 5-1. The state of the rewind flag (table 3-5) is ignored. If the crunch flag is set, the disk space between the current end of the file and the end of the disk area containing it is de-allocated.

A file that is closed with crunch cannot be subsequently extended.

Any close that converts a new file into an old file must supply a file id (or possibly two if indexed) in the FPB, a file type (see table 3-2), an interpreter hardware type (if the file type is interpreter) and

a generation number (if the relevant FPB flag is set (see table 3-4). Closing an old file with this flag set increments the DFH generation number without reference to the FPB. Any close with lock requires an FPB file-id, which overwrites the DFH one. Any close must supply creation and last access dates if the relevant flag is set (see table 3-4).

If a duplicate file condition occurs when a task attempts to close file with lock, the operator has the option of discontinuing the task or deleting the existing old file.

While a disk file is open with output capability, any extensions to a file in the form of records written to file areas that were not previously allocated, are recorded in the FIB. At close time details of newly allocated areas, or a change in the value of the EOF pointer, are transcribed into the file header. This action takes place on every variant of close that does not result in the immediate deletion of the file.

All closes cancel any OTHERUSE restrictions against OPENing by someone else (and stimulates retries by any task waiting for file assignment etc.), but a half closed file remains in (input) use so it cannot be OPENed with LOCK by someone else.

Note that if an old file is affected by REMOVE or PURGE, but other users have the file open or half closed, the file is not deleted from the disk directory. Instead the file is marked as temporary in its header and the file's entry in the directory name list is altered to reflect the file's new temporary status. If a file is closed with purge, or a temporary file is closed with release, then, if there are no other users, the disk space corresponding to the file is cleared and returned to the available pool. The appropriate entries in the directory header list and directory name list are set to available.

CLOSE with REMOVE behaves as CLOSE with LOCK (requiring operator intervention) if new and old file types differ, or if the program could not directly purge the old file. The legality of CLOSE with PURGE or LOCK is shown in table 4-5. Any attempt to alter the directory when the file is called 'SYSMEM' or has the SYSMEM reserved file type is fatal.

The syntax of CLOSE for indexed files is identical to that of the sequential organization files. The two component files of an index file are inseparable and therefore, if the two components are being used as an index file, a single CLOSE applies to them both. CLOSE variants such as PURGE or LOCK apply to both components. Usually, however, one may be temporary and the other permanent, so the action performed on the two files may differ. If required by close, the file id of the key-file must be supplied in the normal file id of the FPB, and the file id of the data file in the second file id. If, while the file is open, the records have been added, the index entries for these records appear in the overflow region of the file. This overflow region must from time to time be consolidated into the index. This process of consolidating the overflow region involves complete rebuilding of the key file and therefore cannot be performed while anyone is accessing the file.

RECORD ORIENTED COMMUNICATES

RECORD ORIENTED COMMUNICATES – SYNTAX

The communicate S-operator is associated with a CPA formatted as below:

```
BYTE 0 VERB   = values
BYTE 1 OBJECT = DST index of FIB
```

Format of remaining bytes depends on verb value

```
#80 < Verb < # 8F
```

If file organization = sequential and ((accessmode = random and verb >81) or (accessmode = sequential and verb = START)) then:

BYTE 2,3,4 – relative record number of required record. (20 bit binary number right justified in 24 bit field)

or if file organization = sequential, access mode = not random and device = line printer (or line printer substitute) then:

BYTE 2	– spacing information
bit 0	= act on paper control information only
bits 2,3	= 11 line advance
	= 10 vertical feed
	= 01 line feed
	= 00 form feed
bit 7 set	= advance before printing
reset	= advance after printing
BYTE 3	– number of lines to advance (0-255) or channel number for vertical tab (0-12)

If file organization = indexed and verb = START or file organization = indexed, access mode = random and verb = READ then

BYTE 2	– record key specifier
bits 0 – 5	set always
bits 6 – 7	= 11 READ equal, START equal
	= 10 READ next, START greater than or equal
	= 01 READ next, START greater than
	= 00 READ next, START next

else no further CPA parameters.

90 < verb < #99

BYTE 2	– DST index of new work area
BYTES 3,4	= offset of new work area within named segment
BYTES 5,6	– length of new work area

Verbs in the range #96 – #99 have no further parameters.

Verbs in the range #90 – #95 have further parameters as detailed below.

Verb = #92 or #93.

BYTE 7 bits 0-3 FUNCTION
bits 4-7 PARAMETER

FUNCTION = 8 – KEYBOARD

PARAMETERS:

bit 5 set	= Acceptable input is command key only
reset	= Total input length in bytes is equal to work area length

bit 6 set	= Numeric keys only
reset	= Alphanumeric
bit 7 set	= Sign allowed
reset	= Sign not allowed

BYTE 8 – if bit 6 is set then this byte exists and contains the number of places to the right of the decimal point.

FUNCTION = 4 – ALARM

FUNCTION = 2 – SET INDICATORS

PARAMETERS:

bit 4 set – bank 0 indicators field present
bit 5 set – bank 1 indicators field present
bit 6 set – bank 2 field present
bit 7 set – bank 3 field present

BYTE 8 et seq. – bank indicator values

FUNCTION = 1 – CLEAR BUFFER

Verb = #90, #91, #94 or #95.

BYTE 7 bits 0-3 FUNCTION
bits 4-7 PARAMETER

FUNCTION = 8 PRINTER

PARAMETERS:

Bit 4 set	= New format CPA
Reset	= Old format CPA
Bit 5,6	= 11 – invalid (ignored)
	= 10 – print(display) in next tab column
	= 01 – print(display) in column given in byte 9 (or bytes 9 and 10)
	= 00 – print(display) in current column
Bit 7 set	= Use underline or alternate ribbon if possible
Reset	= Desist from above
	BYTE 8 – spacing information
bit 0	= act on paper control information only
bits 2,3	= 11 line advance
	= 01 line feed
bits 4,5,6	= 0 – advance all tractors
	= n – advance tractor n (1 <n<7)
bit 7 set	= advance before printing
	reset = advance after printing

If oldstyle CPA then

BYTE 9 – column number (0-255) if parameter bits 5,6 = 01
BYTE 10 – number of lines to advance (0-255)

else

BYTE 9,10 – column number if parameter bits 5,6 = 01
BYTE 11 number of lines to advance (0-255)

FUNCTION = 4 TRANSPORT (Verbs #94, #95 only)

PARAMETERS

bit 4 set = clear head
reset = restore head

FUNCTION = 2 TEST COLUMN (Verbs #94, #95 only)

NO PARAMETERS

FUNCTION = 1 READ-WRITE NUMERIC (Verbs #90, #91 only)

PARAMETERS:

bit 4 set	= sign allowed
reset	= no sign allowed
bits 5,6,7	– as for function 8 above
BYTE 8	– spacing information
BYTES 9,10	– column number if parameter bits 5 and 6 = 0
BYTE 11	– number of lines to advance
BYTE 12	– places to right of decimal point

Verb = #9A

BYTES 2-4 – Binary Sector Address

Verb = #9C

No further parameters.

#A0 < verb < #AF

These verbs map onto the corresponding verbs in the range 80-8F and are used for detecting BLOCK-LOCKED for shared files.

WORK AREA REDEFINITION SEMANTICS

A new work area is defined by supplying its data segment table index, offset within the named data segment and length in the CPA of certain communicates. If the sum of offset and requested length exceeds the length of the data segment holding the work area, or an attempt is made to map the work area onto an inaccessible segment (for example, an FIB) or onto a segment which is not in the user's segment table, the communicate returns a fatal error.

In all cases, if an I/O communicate also redefines the work area, the redefinition is done before the I/O action is started.

If the communicate terminates with an abnormal result (for example, permanent error), then any work area redefinition requested will have been performed. If the communicate terminates with a fatal error, then no assumptions can be made as to whether the work area redefinition was performed.

RECORD KEY SEMANTICS – SEQUENTIAL ORGANIZATION

If the file is open with random access, the READ, WRITE, REWRITE, OVERWRITE and DELETE communicates require the specification of a relative record number as argument. For files open with sequential access, this parameter must be specified for the START communicate. READ, WRITE, REWRITE, OVERWRITE and DELETE do not take this parameter for sequential access files.

The relative record number is supplied as a binary number in the range 1 to 1048560 and indicates the logical ordinal position of the record in the file.

RECORD KEY SEMANTICS – INDEXED ORGANIZATION

If the file is open with random access, the CPA must contain a record key specifier if the verb is READ or START. For all other verbs valid on random access, the record key specifier, if present, is ignored. For files open with sequential access, the record key specifier is only relevant if the verb is START.

For READ on a random access file bits 6 and 7 imply one of two conditions. If the bits = 11 then the value, held in the record work area in the field corresponding to the key of reference is used to retrieve the record. If the bits do not = 11, then the next record in sequence on the key is retrieved.

For START, if bits 6 and 7 = 11, then the file positioned ready to read the first record with key equal to the requested key. If bits 6 and 7 = 10, then the file is positioned ready to read the first record with key greater than, or equal to, the requested key. If bits 6 and 7 = 01, then the file is positioned ready to read the first record with key greater than the requested key. If bits 6 and 7 = 00, then the file is positioned ready to read the next record; that is, START is a no-op.

READ SEMANTICS – SEQUENTIAL ORGANIZATION

READ is valid for files that are open with MYUSE input or input/output and makes a new record visible to the S-program. If the required record is not present in one of the buffers, it ensures that the block containing the required record is queued for input, and puts the task into await state pending the completion of the physical transfer. When the task reawakes, or if the required record is already present, the record is copied from the buffer into the work area supplied by the task. Magnetic tape and cassette files can have a short record at the end of a block. When reading such a record, only the usable information should be transferred to the work area.

The strategies adopted for refilling the buffers depend on ACCESSMODE, MYUSE and, in some cases, the immediate history of the file.

Sequential Access

For sequential access files the target is to keep all available buffers full with records that will be needed subsequently. Usually the buffers are queued for input separately as each one becomes available. However, in some circumstances, such as following an OPEN or START, more than one buffer may be simultaneously available. In this case all the free buffers are queued for successive blocks of the file.

If the file is open input only, then the buffer is declared to be free and is, therefore, a candidate for refilling as soon as the last record in the buffer is copied to the work area.

If the file is open input/output, then the buffer is not declared to be free at this time since it is possible that a subsequent REWRITE may need to reference the same record. The buffer is not released until the current record pointer moves out of the buffer completely; that is, on the first access to another block. In the case of input/output files it is necessary to keep a flag indicating whether any record

in the block has been updated. If the block has been updated, then it must be output before the buffer is released for refilling.

Random Access

In general, for random access files the only buffer that is not marked as empty is the buffer containing the current block of the file. However, this situation may be modified by the READ communicate handler deducing that the file is being accessed sequentially.

If the requested record is not in the same block as the current record, and the key of the requested record is one greater than that of the current record, then the strategy outlined above for sequential access files is adopted. If the key of the requested record differs from that of the current record by a value other than one, then no physical I/O is initiated other than that which may be necessary to input the required record and to output an updated buffer which no longer contains the current record.

On ICMD files, READ always results in the complete record requested being placed in the work area. This always consists of 129 bytes; the first byte is the data mark, the other 128 bytes are the data. All 129 bytes are passed back even if parity is detected, in this case with the usual FCM value (#203010). Care must be taken on ICMD files to skip any defective tracks on the medium.

READ Semantics – Indexed Organization

If the file is open for sequential access, READ causes the next record according to the current key of reference to be made available in the work area. "Next" implies that any deleted records are skipped, and records which are recorded in the overflow region are presented in appropriate key sequence among the non-overflow records. In the event of a duplicate key value being recorded in the key file, then these duplicates are presented in the order in which they appear in the key file; that is, index entries before overflow entries and within each group by logical position. The end of file condition is signalled in the same manner as for sequential organization sequential access files. The end of file condition is given on a read that attempts to access a record sequentially that does not have an entry in the index. If an end file condition has been given, subsequent READs are invalid unless preceded by a successful START or a successful CLOSE, followed by a successful OPEN on this file.

If the file is open for random access and the key specifier in the CPA indicates "next", the external actions of READ are precisely those that would have occurred if the file had been opened with sequential access and the value of the current record pointer had resulted from an earlier sequential READ. If the file is open for random access and the key specifier in the CPA does not indicate "next", then READ causes the first record in the file which contains, in the field corresponding to the prime key of reference, a value equal to that in the corresponding field in the record work area. If a record which has been deleted via another keyfile is accessed, it will be found, although containing all #FFs. If the key is binary zero or contains a byte of #FF, or if a record which matches the requested key cannot be found in the keyfile, then invalid key is signalled in the fetch message in the usual manner (#2030). Note that if an attempt is made to use READ NEXT when the current record pointer indicates the final record in the file, then end of file is returned (#1000). When end of file has been given, subsequent READ NEXTs are invalid until the execution of a successful start, random read or close and open.

INVALID KEY (#2030) is returned after the READ of a sequential access file or READ NEXT on a random access file if the current record pointer is undefined; for example, after an unsuccessful START.

READ SEMANTICS – CONSOLE ORGANIZATION

READ KEYBOARD

Each input string from the keyboard is terminated by a command key. Only those command keys that have been enabled by a set indicators command and those that are permanently enabled (those without associated lights) are valid terminators. If the input string is not validly terminated before the maximum input length (that is, the work area length is exceeded) or, where appropriate, before the places to the right of the decimal point specified in byte 8 are exceeded, then the input is not accepted and an error indication is shown on the console. Note that although typing ahead is allowed, no error indication can be given until the appropriate read command has been issued.

If an error indication is given, then pressing the reset key restores the system to the state it was in immediately before the keystrokes that caused the error. This results in the loss of any input that had been typed ahead. Pressing the reset key when an error has not been indicated clears the input buffer beyond the last terminator. The whole of the input field requested by the read must then be retyped. If no unterminated characters had been input when the reset was keyed, then reset has no effect.

The command key used to terminate the input successfully is returned in status key 2; that is, the third byte of the fetch message. Characters input from the keyboard are not echoed.

For alphanumeric input the characters are moved to the work area, left justified with right space fill. For numeric input the characters are moved to the work area aligned so that the offset from the position of the assumed decimal point to the end of the work area has the specified value. The decimal point itself is not transferred to the work area and is therefore not included in the maximum input length. If a decimal point is not input, the input string is assumed to represent an integer. If the CPA specifies no places after the decimal point, a trailing decimal point can be entered or omitted at will. If a sign is specified as being allowed, then if a sign character is input, the corresponding ASCII sign character is placed in the first byte of the work area. Both the minus key and the reverse entry key produce the ASCII minus character. If no sign character is keyed in but a sign is allowed, then the first byte of the work area is set to the plus character. Allowance for the sign character must be made in the total input length. All bytes in the work area that do not correspond to keyed in characters (except sign) are filled with the ASCII zero character.

Alarm

The console alarm is sounded once.

Set Indicators

The 64 possible indicators are divided into two banks with:

BANK 0 – 01 to 32
BANK 1 – 33 to 64

For each bank field specified as present by the corresponding bit in the parameter field there is a four byte group in the CPA. Each bit in the group corresponds to an indicator within the corresponding bank (most significant bit = lowest numbered indicator). Each bit takes the value 1 for indicator on, or 0 for indicator off. When a READ KEYBOARD or READ-WRITE KEYBOARD/PRINTER communicate is performed, the indicators specified in the last SET INDICATORS communicate received since the last READ or READ-WRITE are illuminated. If no SET INDICATORS communicate was received since the last READ or READ-WRITE, then no indicators are specified. The specified indicators remain illuminated from the time the READ or READ-WRITE is issued until the keyboard input

message is successfully terminated. The indicators then remain extinguished until a READ or READ-WRITE is preceded by a SET INDICATORS command.

There are two other banks that do not correspond to indicators:

BANK 2 – 65 to 96
BANK 3 – 97 to 99

For each of these banks, a four byte group, similar to those described above, can exist in the CPA. A bit set in one of these groups does not cause any indicators to be illuminated but does cause certain keys to be enabled.

CLEAR BUFFER

On receipt of this command no further key strokes are accepted from the keyboard. The error light is illuminated and the audible alarm sounded. Any characters which are buffered ahead are cleared from the keyboard buffer and only the reset key may be pressed. When the reset key is pressed, the error light is extinguished, characters are once more accepted from the keyboard, and buffering is resumed to allow typing ahead.

Note that a reset key which has been typed ahead does not satisfy this communicate; it insists on the reset key being pressed after sounding the alarm as an acknowledgement from the operator that the error condition has been noticed.

WRITE, REWRITE, OVERWRITE, SEMANTICS – SEQUENTIAL ORGANIZATION

REWRITE and OVERWRITE are only valid for disk files open with MYUSE input/output. If ACCESSMODE is sequential, WRITE is only valid for MYUSE output. REWRITE is only valid if the previous operation on this file was a successful READ. OVERWRITE is invalid if the previous operation was a START or OPEN. If ACCESSMODE is random, WRITE is valid for MYUSE output or input/output.

All three communicates cause the copying of the current contents of the work area to the appropriate position in the buffer and, if necessary, initiate the physical output of the block. For random access blocked files these operations are preceded (if necessary) by a physical input operation in order to establish correctly the other records in the target block.

For random access files the position which the record occupies in the file is determined by the record key parameter in the CPA. The semantics of all three communicates, when applicable, are identical for random access files. INVALID KEY (#2040) is returned if the record specified is greater than the original declared size of the file.

For sequential access files the key value is determined implicitly as follows:

1. WRITE

Implicit key is one higher than the implicit key of the previous operation. When a disk file is opened with otheruse shared, the previous operation on the file may have been done by a different task.

2. REWRITE, OVERWRITE

Implicit key is the same as the implicit key of the previous operation.

If the device is a disk, for both the random and sequential case, INVALID KEY (#2040) is returned if WRITE attempts to add a record which would make the total number of data records in the file greater than its original declared size.

If the device is a line printer or a line printer substitute (for example, a console printer), then bytes 2 and 3 of the CPA contain paper spacing information. Byte 2 specifies four functions; line advance, vertical tab, line feed and form feed. Line advance implies that the paper is advanced vertically by the number of lines specified in byte 3. A line advance value of zero is permissible. The line feed function does not take any other parameter and is equivalent to a line advance with a value of one. The vertical tab and form feed functions are only relevant to devices such as line printers equipped with format control tapes or similar vertical sensing devices. For the semantics of these commands when sent to a console printer, refer to the FD description in the SCL specification. Vertical tab causes the paper in the channel of the format tape specified in byte 3. Form feed causes the paper to be positioned at the top of the next page (assuming the presence of a correctly punched and aligned format tape).

If the device is an ICMD, INVALID KEY (#2040) is returned if WRITE attempts to add a record beyond the physical capability of the device. The record for a WRITE to ICMD consists of 129 bytes; the first is the data mark, and the other 128 are the actual data.

If an error is detected on the write, or subsequent read after write check, then an attempt is made to write a "physically deleted" control record to the faulty sector.

If this indication can be written successfully, then abnormal condition (relocation successfully attempted) is returned (#3050). If the control record cannot be successfully written and verified, then abnormal result, write error (#3020) is returned. Care must be taken, on an ICMD, to skip any defective tracks.

REWRITE, OVERWRITE SEMANTICS – INDEXED ORGANIZATION

REWRITE and OVERWRITE output the record in the work area to the file in the position indicated by the value in the key field of the work area. The communicates are unsuccessful and return an INVALID KEY (#2030) fetch message if;

the key is binary zero or contains any byte of #FF
or if a record with that value of key did not already exist in the file
or if it has been deleted since a preceding read or start and fail fatally if the file is not open with MYUSE = I/O.

Overwrite is invalid and fails fatally if the key file is a null key file.

For files open with sequential access the I/O operation on this file immediately preceding a REWRITE must have been a successful READ, or else a fatal error ("bad sequence") is given. The value of the key presented by the REWRITE must be the same as that returned by the read or else an INVALID KEY (#2010) return is given.

The semantics of REWRITE on random access files, and of OVERWRITE on both random and sequential access files, are identical. The communicate can be preceded by any valid I/O communicate or START or OPEN. The value of the key can be that of any pre-existing record in the file. If the key value corresponds to the current record pointer, then the contents of the work area are output to that record. If the key value does not correspond to the current record pointer, then the contents of the work area are output to the position of the earliest record in the file with that key value; that is, effectively at random.

Note the effect if the rewritten or overwritten record is one of a group of duplicates. If the record has the same key as the record indicated by the current record pointer (in general, the last record read)

then the current record is updated. If the new record has a different key value, then the record rewritten is the first (oldest) record of the set of duplicates with that key value.

The action of the communicate does not modify the value of the current record pointer. Therefore, in an alternating sequence of sequential reads and overwrites, or random rewrites, the reads access successive records in the file, irrespective of the positions to which the overwrites or rewrites direct the records.

WRITE SEMANTICS – INDEXED ORGANIZATION

WRITE outputs the record in the work area to the position indicated by the value in the key field of the work area. Unless duplicates are allowed, the communicate is unsuccessful and returns an INVALID KEY (#2020) fetch message, if a record with that value of key already exists in the file. If duplicates are allowed, the key is inserted after any existing keys of the same value, and this preserves the order of insertion of records.

A fatal error is given if the key value supplied is binary zero or contains any byte of #FF.

If the file is open for sequential access, WRITE is only valid if MYUSE = output. Any other value causes a fatal error. The value of the key supplied with each record must be greater than, or equal to, that supplied in previously written records. If it is not, the WRITE is unsuccessful and returns an INVALID KEY (#2010) fetch message. Note that the validity conditions for sequential access files of WRITE and START are mutually exclusive.

If the file is open for random access, WRITE is valid for MYUSE = output or input/output. The value of the key supplied in the work area can take any value that does not violate the constraints of the first paragraph, and the record will be positioned appropriately in the file index.

For both random and sequential access, INVALID KEY (#2040) is returned if WRITE attempts to add a record which would make the total number of data records in the file exceed the maximum number of records that were declared when the file was created. Note that if a data record is deleted, although it is no longer accessible, it still probably occupies space in the key file and the data file, and is, therefore, counted when computing the total file size. (Deleted key records can be removed at close time. Deleted data records are completely removed by a separate utility.)

In common with all output communicates on indexed files WRITE does not modify the value of the current record pointer. Since the destination of the record is always determined by the value in the key field, it is not necessary to use the value of this implicit record pointer on output operations.

WRITE SEMANTICS – CONSOLE ORGANIZATION

WRITE PRINTER

In the WRITE PRINTER (and READ-WRITE PRINTER, following) communicates, the concept of the record is extended to console files. The concept is best explained in terms of the printer.

Each write (or read-write) carries as a parameter the column at which printing is to commence. The communicate also carries information as to whether to advance the paper or not. A communicate that advances the paper is regarded as terminating the record. Until a communicate to advance the paper is issued, fields within the record (line) can be filled in or overwritten in an arbitrary order. When a line advance has been given, no further updating of that record (line) is possible. The printing associated with the communicate that specifies the line advance can be part of the new record, or part of the old record, depending on whether the advance is specified to occur before or after printing. The maximum value of record size is the available width of the printer, but can be smaller if desired. The

record concept applied to self-scan is similar, except that the maximum record size is the full screen size rather than the line size.

The characters in the work area are printed or displayed, starting in the column specified by the parameter field. Where the output device is a self-scan display, the record size, and therefore the maximum permissible column number may be greater than the screen width. Column number in this case is interpreted as (integer quotient of column number and screen width) complete lines plus (remainder) columns from the start of the record. Each record starts on a new screen line.

Setting the underline flag in the CPA causes the characters to be underlined (if the output device supports inplace underline) for this write operation only.

Three variants are permitted in the specification of column numbers;

1. Position to current logical position; for example, to the end of the last printed field.
2. Position to an absolute column number relative to the start of the record.
3. Position to the next tab position. The tab positions are N columns from the start of the record, where N is the lesser of 40 or the screen width. However, if the next tab position would be beyond the record boundary, then the command is interpreted as line feed and position zero; that is position to the start of the new record.

The semantics of format control differ, depending on whether the output device is a self-scan or a printer.

1. Printer

Depending on the value placed in the spacing information byte, the paper is advanced before or after printing. The spacing information byte specifies two functions; line advance and line feed. Line advance implies that the paper is advanced vertically by the number of lines specified in byte 10 or 11. A line advance value of zero is permissible. The line feed function does not take any other parameter and is equivalent to a line advance with a value of one.

2. Self-scan

The format control for self-scan is essentially the same as already described for printer. However, the following points should be noted.

The current record is displayed at the foot of the screen. Only a sufficient number of lines of the record to accommodate the highest character position within the record that has been referenced by a write (or read-write) are displayed. Unreferenced lines at the end of the record are not displayed, nor are they displayed when the record is scrolled to accommodate a new record.

A line advance of zero causes no movement of the screen image other than that which may be necessary to accommodate a new highest referenced character.

A line advance of one, or a line feed, causes the present contents of the screen to be rolled upwards by a sufficient number of complete screen lines to accommodate the referenced area of a new record at the foot of the screen. A line advance of greater than one causes the above action, plus an additional advance of one screen line, for each requested line advance in excess of one.

If the format control information specifies advance before printing, then it is acted on immediately.

If the format control information specifies advance after printing, then the action is deferred until the next write communicate is issued.

Transport

If bit 4 of the parameter field is SET, then the print head is cleared. If bit 4 of the parameter field is reset, then the print head is restored. A request to clear the head when it is already cleared, or to restore it when already restored, is a no-op.

For self-scan displays, a clear head is interpreted as extinguish cursor, and restore head is interpreted as restore cursor.

Test Column

This returns, in status keys 1 and 2, the number of the next column; that is, the column implied by the parameter bit 6 reset in WRITE or READ-WRITE. However, after READ-WRITE it gives the end of the string actually indexed, rather than the end of field defined by the READ-WRITE.

READ-WRITE SEMANTICS – CONSOLE ORGANIZATION

READ-WRITE PRINTER

The semantics of the format control component of the syntax is identical to that of the WRITE PRINTER communicate. Each character input by the operator is entered to the work area (left justified) and echoed to the printer/screen at the specified position. The input character string is terminated by a command key. Only those command keys that have been enabled by a set indicators command and those that are permanently enabled (those without associated lights) are valid terminators. The use of an invalid command key as a terminator results in an error indication (error light and bell). An error indication is also given if a valid terminator is not input before the field size defined in the CPA is exceeded. Note that although typing ahead is allowed, no echoing or terminator checking can take place until the appropriate READ-WRITE has been issued.

If an error indication is given, then pressing the reset key restores the system to the state is was in immediately before the keystroke that caused the error. This results in the loss of any input that had been typed ahead. Pressing the reset key when an error has not been indicated but a "read-write" is in progress clears the input buffer. The current head position reverts to that indicated by the current READ-WRITE, and the whole input field requested by that READ-WRITE must then be re-typed. If no characters had been input when reset was keyed, reset has no effect.

The command key used to terminate the input successfully is returned in status key 2, and is not transferred to the work area. If the input length is less than the work area length, the work area is filled on the right with ASCII space characters.

READ-WRITE NUMERIC

The semantics of this communicate depend on whether the output device is a printer or a display. If it is a printer, then the semantics are the same as for READ NUMERIC. If it is a display, then the following semantics apply.

When the communicate is issued, the field specified by the CPA is filled with blanks, except for the character position corresponding to the decimal point: a point character is displayed here.

Note that the field size is one greater than the length specified in the CPA, since the latter does not allow for the decimal point which is not transferred to the work area.

The field width makes allowance for the point even if the CPA specifies no characters after the decimal point. In this case the point is displayed as the last character of the field.

The only valid input keys are the numerics, the decimal point, any enabled terminators and, if specified in the CPA, the sign or reverse entry keys.

Any numeric keys input before the decimal point are echoed at the position immediately to the left of the displayed point. Previously displayed characters are shifted left one place at a time to accommodate the new characters. Numeric keys input after the decimal point has been keyed are echoed to the right of the displayed point with normal left to right escapement. If the number entered is an integer, the decimal point need not be entered before terminating the field. The sign character (or reverse entry key) can be entered at any time during the field. It is always echoed at the extreme left of the echoed field. Multiple depression of the sign key is not an error. A second input of a sign character overwrites the first, both on the screen and in the buffer; for example, input of a "+" character countermands a previously entered "−".

Error indications are given if the field sizes to the right or left of the point are exceeded, or a second point character is entered. Indexing reset at this point restores the system to the condition preceding the incorrect keystroke. Indexing a second reset, or resetting when no error indication has been given, results in a restart of the whole read-write operation.

DELETE SEMANTICS

DELETE is only valid for disk files open with MYUSE = input/output. Access mode can be random or sequential.

Deleted in this context implies that the record in the data file is marked as deleted by setting every byte to #FF. If the file in question is an indexed file, the key file entry corresponding to the deleted data record is marked in addition by setting the key value to zero, leaving the logical record number fields unchanged.

The rules governing the selection of the record to be deleted are the same as those pertaining to the REWRITE communicate.

Note especially in this context the rules pertaining to duplicate keys.

Possible fetch messages also correspond to those for rewrite. In common with all output communications, the contents of the work area are not preserved by the delete communicate.

START SEMANTICS – SEQUENTIAL ORGANIZATION

START is only valid for disk files open with ACCESSMODE sequential and MYUSE input or input/output. The argument of the START communicate, specified in the relative key slot in the CPA, gives the implicit key to be used (without incrementing) on the next READ performed on this file.

START SEMANTICS – INDEXED ORGANIZATION

START is valid for files open with MYUSE = input or input/output and ACCESSMODE random or sequential. (Note that ANSI 74 COBOL does not support START on random access files. However, it is necessary for us to support this feature in order that S-programs are able to simulate dynamic access mode by the use of random access mode.)

START is invalid and returns a fatal error if the key file is a null key file.

The semantics of START are the same for both random and sequential access. The action of START is to reposition the current record pointer. The requested key is placed in the field of the work area corresponding to the prime key of reference. Depending on the value set in the key specifier in the

CPA, the current record pointer is repositioned to point to the first record with key equal to, equal or greater than, or greater than, the requested key. If this key is binary zero, or contains a byte of #FF, or if no record satisfying the specified relationship can be found, then INVALID KEY (#2030) is returned in the fetch message and the value of the current record pointer is undefined. If the key specifier value implies "next", then START is a no-op.

GET SEMANTICS

GET is only valid for sequential organization files open with access mode stream and myuse input. Each GET requires the respecification of the work area. As a result of each GET, a number of bytes equal to the work area length are transferred from the file to the work area. Other than the practical limitations on the size of work area, there is no limit on the number of bytes which can be accessed by a single communicate. A single GET can cause several physical I/Os and, if the work area length exceeds the total length of the available buffers, it may require more than one stage of waiting for physical I/O to complete and to copy the filled buffers into the work area. If a GET has a work area length that implies referencing bytes beyond the end of file pointer, then only those bytes up to the end of file pointer are transferred to the work area. The remainder of the work area is filled with binary zeros, and an end of file indication (#1000) is returned in bytes 1 and 2 of the fetch message. Any further GETs, after OEF is given, result in a fatal error.

If the specified work area extends beyond the boundary of the work area segment, a fatal error is returned in the FCM (EVENT 71).

PUT SEMANTICS

PUT is only valid for sequential organization files open with access mode stream and myuse output. The semantics are analogous to those of GET except that the direction of transfer is from work area to file. EOF is given after a PUT if an attempt is made to add bytes to a file that would make it exceed its declared maximum size. In this case only as many bytes as will fit within the declared maximum size are transferred.

If the specified work area extends beyond the boundary of the work area segment, a fatal error is returned in the FCM (EVENT 71).

If the file was a line-printer which was directed to back-up, each record of a block in the printer back-up file formed via PUTs contains printer control information specifying a single line advance.

STREAM CONTROL SEMANTICS

STREAM CONTROL is only valid for sequential organization files on a line printer, or line printer substitute, with access mode stream and myuse output. As a result of each STREAM CONTROL, the physical output of the bytes PUT to the current block is initiated and the spacing control information from bytes 2 and 3 of the CPA is applied.

If the printer file has been directed to backup, the semantics are slightly more complex. The current block is space filled in exactly the same way as would have occurred using PUT communicates, with each record containing control information specifying a single line advance. The control information in the last record of the block contains the information received with the STREAM CONTROL.

CONDITIONAL CLASS A COMMUNICATES

Errors which would cause a fatal FCM in the case of an unconditional class A communicate cause the return of an FCM if the communicate is conditional.

For most unconditional class A communicates there is a probability that the task will be suspended before the communicate is completed. For example, a READ may be waited pending the physical I/O needed to fill the buffers.

The circumstances under which a conditional communicate may lose control are generally limited to waits for relatively rapid system activities. The present list of circumstances under which a conditional communicate may lose control is:

1. during virtual memory transfers; for example, for a record work area in an overlayable data segment.
2. during disk allocation; that is, while waiting to access the available table for example, but not while awaiting operator action if allocation is unsuccessful.
3. during an automatic change reel on a magnetic tape file.

Note that case 3 can result in an arbitrarily long wait while the operator mounts the continuation reel. If it is important that the program is never waited for operator intervention, the program should avoid using multireel tapes.

At any point (other than those above) at which an unconditional communicate would lose control, the conditional communicate restarts the user with a fetch value indicating conditional failure ('resource temporarily unavailable'). With the exception of conditional failures caused by inability to find available disk space, conditional communicates automatically converge towards success. If there is only one point at which an unconditional communicate might lose control for reasons other than those listed above, then the conditional variant of that communicate issued after the failure of an identical conditional communicate will always succeed. This will occur provided that sufficient time elapses between the two communicates for the completion of the necessary I/O. If the second communicate is issued too soon it gives a conditional failure like the first and is otherwise treated as a no-op; that is, it will not repeat the initiation of asynchronous I/O performed by the first communicate. Certain unconditional class A communicates may lose control for reasons other than those listed at more than one point. For example, READ on an I/O file may have to wait first while outputting an updated buffer and then while refilling the buffer. The actions performed by the communicate handler between the two physical I/Os cannot be performed by the interrupt state machine. They are most conveniently performed by the task which requires the I/O, and therefore the second physical I/O is not initiated until the task issues a communicate after the completion of the first physical I/O. Therefore, in such a case, the task receives at least two conditional failures before achieving success.

The case of conditional failure due to the lack of disk space deserves special note. Although the lack of disk space is notified to the operator by the usual SPO message, there is no guarantee that the operator will take any action to make more space available. Therefore, there is no assurance that a reissue of the failed conditional communicate will succeed, even after an arbitrary elapse of time. For this reason, the conditional failed fetch message distinguishes the 'no user disk' condition from other failure conditions by placing the appropriate message number in bytes 1 and 2 of the fetch message.

The conditional variant of the class A verb is distinguished from its unconditional counterpart by verb value. The verb value for each unconditional verb is an even number; the verb for the conditional variant is one greater. Conditional variants are defined for all class A verbs except those from stream I/O.

With one exception the external consequences of the failure of a conditional communicate are as if the communicate had never been issued. The failed communicate can be followed by any other class A communicate that would be valid following a successful communicate or by close. However, it must be noted that although it may be valid to follow a failed conditional communicate by a communicate that differs in verb or record number, it may not be efficient, since the internal actions necessary to

ensure convergence to success are likely to disrupt any buffering ahead that has been performed. The exception to this rule is READ on a sequential access I/O file. A failed conditional READ on a sequential access I/O file cannot be followed by any communicate implying output.

Certain unconditional class A communicates (for example, START) never lose control. Therefore, there is no difference between the conditional and unconditional variants of these verbs. However, if it is vital to the logic of the program that such a verb does not lose control, then the compiler generates the conditional variant since no guarantee can be offered that the unconditional variant will never lose control in subsequent implementations of the operating system.

Special note must be made of the verbs in the range #A0 – #AF. These are treated as unconditional in all cases except in the detection of a BLOCK-LOCKED conditional. In this case an abnormal result (#9020) fetch message is returned.

There are no conditional variants for these communicates.

If a BLOCK-LOCKED condition is encountered on a conditional communicate, bytes 1 and 2 of the fetch message contain a conditional error (#9020).

TEST STATUS

Certain users have a requirement to verify whether a file has queued I/O outstanding. For example, a user can perform a disk write communicate and at some later stage in the program may need to confirm that the disk file has been updated. He may also need to wait until it has been updated before proceeding. If the file has only one buffer, this can be achieved by performing a conditional write (or an unconditional write if waiting is required), but this is unsatisfactory since it involves initiating a redundant write and does not cover the multiple buffer case.

The test status communicate is provided for this purpose and causes all the records which have been logically written to be physically copied to the media. If the file is open SHARED, all updated buffers are physically copied, regardless of which shared user(s) updated them.

The communicate is only valid in the following cases:

1. Sequentially organized, sequential or random access disk files.
2. Indexed organization, sequential or random access disk files. In this case only the data file records are tested.
3. ICMD files.

Any other application of this communicate results in a fatal error.

The unconditional variant returns three possible values in the fetch message waiting for I/O complete, if necessary:

1. #00 00 00 if all I/O buffers for the file have been completed successfully.
2. #20 30 00 if all I/O buffers for the file have been completed, but not all are successful.
3. #20 10 00 if all I/O buffers have been completed but one or more have an EOF indication. Note response 2 takes precedence over response 3.

The conditional variant is different only in that it does not wait for completion of the physical I/O transfers which it initiated. Therefore, one more fetch value is required:

4. #40 00 00 if one or more I/O buffers for the file have not yet been completed.

The communicate does not distinguish between multiple errors, and produces no output message. Distinguishing individual errors and printing appropriate messages does not take place until the buffer is subsequently accessed by a normal I/O communicate.

FREE BLOCK

This communicate unconditionally frees the block locked by the issuing task/FIB pair. If no block is locked, the communicate is treated as a no-op. No error conditions can occur.

FIELD ORIENTED COMMUNICATES

DISPLAY/ZIP/PAUSE

DISPLAY/ZIP/PAUSE – Syntax

The communicate S-operator is associated with a CPA having the following format:

BYTE 0 VERB = # 10 to # 1F

#10 = undefined
#11 = zip
#12 = display
#13 = zip and display
#14 = pause
#15 = zip and pause
#16 = display and pause
#17 = zip, display and pause
#18 = undefined
#19 = undefined
#1A = conditional display
#1B = zip and conditional display
#1C = display without logging
#1D = system display
#1E = undefined
#1F = undefined

BYTE 1 OBJECT = DST index for test area

(BYTES 2, 3, 4, 5 ADVERB)

BYTES 2, 3 = 16 bit offset into object segment for text area

BYTE 4, 5 = length of text in bytes

DISPLAY/ZIP/PAUSE – Semantics

Several independent functions can be invoked in combinations controlled by the value of the less significant digit of the verb. All possibilities are valid, although certain values are currently undefined. Multiple functions are executed in the following sequence: display, zip, pause. Display (if requested) may be conditional.

The independence of the bits in this less significant digit breaks down with verbs of #1C and #1D.

If the verb is #1D (system display), the display is directed to the system SPO. Any other display is directed to the originating SPO.

PAUSE without ZIP suspends the task until the operator restores it to execution.

If the verb is #14 (pause) the object and adverb are not examined (and need not exist). Format characters for displayed messages are inserted automatically by the virtual machine. Unless the appropriate bit is set in the PBB priority class field, the displayed text is prefixed by the task's mix number and name.

A display may have carriage returns/line feeds inserted by the system. When this happens, the continuation line is preceded by three dots in the first three character positions. These first three positions are not available to user programs; a carriage return character in the display text always positions the print head or cursor at column four.

The text string may also contain the line feed and bell characters. If the string contains any of these characters, and is also to be used as a ZIP text, it should be noted that these characters will not be stripped from the initiating message before it is passed to the program.

The display without logging variant (#1C) defaults to the normal display (#12) if the task does not have SYSMEM privilege (bit 1 of the priority class in the PPB).

A zip is indicated by verb values #11, #13, #15, #17 and #1B and is used to initiate another task or do other SCL. The format of the zip text should be the same as the format of an SCL message which would initiate the functions from the SPO. The zip text is not subject to the same restrictions on characters set and total length as SCL input. A zip text can be up to 701 bytes in length and contain any binary values. If the attempt to load the requested program fails for any reason the appropriate error indications can be found in the fetch message following the completion of the zip. Similarly, the failure of any zipped SCL intrinsic is signalled in the fetch message. In either case, byte zero of the FCM is set to #20.

If the failed zip communicate was a combined zip and display (verbs #13, #17 and #1B) a fail message is printed. This message is the one which would have been produced if the SCL had come from the keyboard.

The variants on this communicate which specify a ZIP cause the initiating task to be suspended until the SCL intrinsic is completed. If the zip text was a request for the initiation of another task and bit 5 is set (that is, PAUSE), then the initiating task is suspended until the initiated task runs to end of job (or enters a SUPER.ACCEPT wait in the case of superutility).

If a task has been initiated by a zip with PAUSE, then there is a lasting connection between this task (the son) and the initiating task (the father). If the son is DS'ed or DP'ed, an abnormal result is passed back to the father. If the father is DS'ed or DP'ed the son immediately suffers the same fate. A task initiated by a ZIP without PAUSE is an entirely independent entity.

If the communicate is a ZIPPED superutility request and superutility is not in a superaccept wait, the system automatically re-issues the superutility request, without informing the task which issued the communicate, until success is ultimately achieved.

ACCEPT

ACCEPT – Syntax

The communicate S-operator is associated with a CPA of the following format:

```
BYTE 0 VERB = #20
BYTE 1 OBJECT = DST index for text area
```

BYTES 2, 3 = 16 bit offset into the object segment for text area
BYTES 4, 5 = maximum length of text in bytes

ACCEPT – Semantics

The virtual machine advises the operator that input is expected and suspends the task until the operator responds. The operator may not provide input in anticipation of an ACCEPT communicate. The input is transferred into the text area, left justified, with a trailing blank fill. Input beyond the maximum length specified will be truncated. The input is echo printed.

MISCELLANEOUS COMMUNICATES

DATE/TIME

DATE/TIME – Syntax

The communicate S-operator is associated with a CPA formatted as follows:

BYTE 0 VERB = #40
BYTE 1 OBJECT index into user's DST
BYTE 2,3,4 ADVERB
BYTE 2,3 16 bit offset into segment indexed by object
BYTE 4

bit 0,1 = 00 data as YYMMDD (B C D)
= 01 date as YYDDDD (B C D)
= 10 time as HHMMSS (B C D)
= 11 time as binary number of tenths of a second

DATE/TIME – Semantics

If the time facility is not implemented on a particular implementation, the communicate writes 240000 in BCD numbers or an equivalent binary number of tenths of a second into the specified address.

TERMINATE

TERMINATE Syntax

The communicate S-operator is associated with a CPA formatted as follows:

BYTE 0 VERB = #41
BYTE 1 bit 0 set = pass 2 bytes back into father's fetch message
bit 1 set = lock VM file
BYTES 2,3 = message for father if bit 0 of byte 1 is set (placed in bytes 1,2 of father's fetch message)

TERMINATE Semantics

The TERMINATE communicate is used to signal end of job, and results in the dismantling of the task's run structure.

If the task was initiated by a ZIP with PAUSE, then when this task (the son) terminates, the initiating task (the father) is restarted. If the son's terminate communicate has bit 0 of byte 1 set, the contents of bytes 2 and 3 of the CPA are moved into bytes 1 and 2 of the father's fetch message. Note that

there is no ambiguity between the son's terminating message and a possible ZIP failure message; the former has byte 0 of the fetch message set to zero and the latter has byte zero = #20 (that is, abnormal result).

If the terminating task was not initiated by a ZIP with PAUSE, any message for the task's father is ignored.

If bit 1 of byte 1 on the CPA is set, the virtual memory file for the task is locked (that is, inserted in the permanent disk directory) with name DMFIL<MIX>, where <MIX> is the terminating task's id as described in the SCL specification. If the file already exists on the disk with this name, the usual duplicate file message is given. The operator then has the option of removing the earlier file with that name and continuing the present task which will then terminate in the required fashion, or discontinuing the present task with a 'DS' message which causes the task to terminate without locking the task file. Note that discontinuing the task with a 'DP' message is equivalent to continuing it in these circumstances.

WAIT

WAIT – Syntax

The communicate S-operator is associated with a CPA formatted as follows:

BYTE 0 VERB #42
BYTES 1,2 16 bit binary number of seconds

WAIT – Semantics

This communicate suspends the task, and it is restored to execution after the specified number of seconds have elapsed.

COMPLEX WAIT

COMPLEX WAIT – Syntax

The communicate S-operator is associated with a CPA formatted as follows:

BYTE 0 VERB = #44
BYTES 1,2 = 16 bit binary number of seconds
BYTE 3 = Flags:
Bit 0 – set => NOCLOCK option has been specified
Bits 1-7 – reserved and reset
BYTES 4-10 = Event class indicators, in priority order
Each byte contains 2 event class values, the first digit of zero terminates the list
BYTE 11 = DST index of response area
BYTES 12, 13 = 16 bit offset into object segment for response
BYTES 14,15 = maximum length of response area in bytes

Response Area Format

BYTES 0,1 EVENT CLASS NUMBER
BYTES 2-(LENGTH – 1) ADDITIONAL INFORMATION

The event class number identifies the class awaking the task. It contains values defined as follows:

The additional information field is dependent on the event class returned:

EVENT CLASS	ADDITIONAL INFORMATION	BYTES
0(Timeout)	None	
1(MCSQ)	None	
2(SCLQ)	None	
5(SUBQ)	Name of awaking Subq.	2-13

COMPLEX WAIT – Semantics

The communicate allows the requestor to wait until one of a list of events occurs, specified by the requestor in priority order in bytes 4-10 of the CPA. These events are:

EVENT CLASS	VALUE
MCSQ	#1
SCLQ	#2
SUBQ	#5

If an unknown event is specified, an abnormal result (#9030) is returned in the fetch message.

In addition, a TIMEOUT event class, value 0, is always specified by the value in bytes 1 and 2 of the CPA. This event class causes the issuing task to be restored to execution when the specified time has elapsed. However, if the NOCLOCK bit is set, the given time value is ignored and the requestor will never be restored to execution by this event class.

MCS QUEUE (MCSQ): This event class causes the requestor to be restored to execution when a message is present on that task's MCS queue. A wait of this type will only by issued by an MCS.

SPO MESSAGE QUEUE (SCLQ): This event class causes the requestor to be restored to execution when a message is present on that task's message queue. If the CPA contains this event class and the requestor is not controlling remote system operation, an abnormal result (#9040) is returned in the fetch message.

SUBNET QUEUE (SUBQ): This event class causes the requestor to be restored to execution when a message is present on any subnet queue that is attached to the task. In addition, the name of the subnet queue causing this event will be returned in the response area. If the requestor has no subnet queues attached when the communicate is issued, or if all subnet queues become detached while the requestor is waiting, an abnormal result (#9050) is returned in the fetch message.

The communicate is processed as follows:

1. If the NOCLOCK bit is reset and there is no clock available to the virtual machine, the communicate returns immediately, indicating the success with the response area containing an event class number of @0000@, and indicating timeout.

2. Each class within the event list is then examined individually from the first entry. If an illegal class, or the occurrence of an event is detected, the communicate returns the appropriate fetch message and response area.

3. After the entire list has been examined.

1) If the NOCLOCK bit is reset, the virtual machine starts its time.
2) The requestor is suspended.

4. The requestor remains suspended until the occurrence of an event, as defined by the event class. The virtual machine then reports success or failure in the fetch message and the appropriate event class and additional information in the response area. The requestor is then restored to execution.

Note that the initial scan of the event list in step 2 must ensure that no event goes unreported because it occurs after its entry has been examined, but before the task is suspended.

If more than one event occurs before the requestor can be restored to execution under step 4, the highest-priority event must be reported in the response area.

COMMUNICATES FOR REMOTE SYSTEM OPERATION

Immediately following system start-up, the system interface device will be the local interface device, if there is one. If no local interface device exists, then system output will just be logged. This situation prevails until a program requests to handle remote operation and the MCP approves the request.

Program to MCP communication is via the normal CMS communicate path – on the order of a zip. There is no input queue to the MCP. As this line of communications is to be the new input SCL path, the existing restrictions on SCL input will be enforced.

MCP to program communication is through a virtual queue. Entries in this queue include the message text and six byte header. This header consists of one byte of flags, one byte of length, two reserved bytes, and two bytes for the event number. As the container size for the length indicates, entries are limited in length. Messages which exceed this limit will be partitioned into the appropriate number of entries with the flag byte indicating continuation. The MCP places all system generated output messages into this queue.

LOG ON

LOG ON – Syntax

The communicate S-operator is associated with a CPA formatted as follows:

BYTE 0 VERB = #50

LOG ON – Semantics

If all conditions are met, the requestor is logged-on as the new task controlling remote system operation, by effecting the new routing path for output messages and creating the virtual queue.

Failure to log-on results in an abnormal result fetch-message being returned. An FCM of #1000 in bytes 1 and 2 denotes that the system has not been logically configured to permit remote operation (denoted in the machine-dependent section of SYSCONFIG). An FCM of #2000 denotes that another program is already logged on for remote system operation.

LOG OFF

LOG OFF – Syntax

The communicate S-operator is associated with a CPA formatted as below:

BYTE 0 VERB = #51

LOG OFF – Semantics

If the requestor is not controlling system operation, an abnormal result (#2000) is returned in the fetch message. Otherwise, the routing of system output messages reverts to that used immediately following system start-up and the virtual queue is erased, causing the loss of any messages in, or being inserted into, it. An implicit log-off is issued by the virtual machine if the program controlling remote system operation goes to EOJ without first issuing a log-off communicate.

RUN

RUN – Syntax

The communicate S-operator is associated with a CPA formatted as below:

BYTE 0 VERB = #52/#53
BYTE 1 = DST index for data area containing the text
BYTES 2,3 = 16 bit offset into the segment for the data area
BYTE 4,5 = Length of data area in bytes

The data area has the following format:

BYTES 0-5 Reserved
BYTES 6-N Text

The purpose of this verb is to supply operator-input to the virtual machine while it is under remote operation.

If the requestor is not the program controlling remote system operation, an abnormal result (#2000) is returned in the fetch message.

If the data area length is greater than 255 characters, an abnormal result (#3000) is returned in the fetch message.

If the verb has been specified as conditional (#53) and the requestor would have to wait for the communicate to be handled, a conditional failure (#0000) is returned in the fetch message. If the verb is unconditional (#52), the requestor waits if necessary for the communicate to be handled.

When the text has been accepted for processing, a success fetch message is returned. Errors in the execution of the text's contents are not returned via the RUN communicate.

READ MESSAGES QUEUE

READ MESSAGES QUEUE – Syntax

The communicate S-operator is associated with CPA formatted as below:

BYTE 0 VERB = #54
BYTE 1 = DST index for data area receiving the text
BYTE 2-3 = 16 bit offset into segment for the data area
BYTES 4-5 = length of data area in bytes

The data area has the following format:

BYTE 0 Flags,
Bit 0 – set => this message is continued in the next queue entry
Bit 1 – set => this message has been truncated due to insufficient text space
Bit 2 # set => the virtual queue has overflowed
Bit 4 – set => no-log display
Bit 3,5-7 – reset – reserved
BYTE 1 Length of queue entry text in characters, 0<length<161
BYTES 2-3 Reserved
BYTES 4-5 Event number, binary
BYTES 6-N Queue entry text

If the requestor is not the program controlling remote system operation, an abnormal result (#2000) is returned in the fetch message.

The continuation of the flag is set if the output message from the virtual machine is continued in the next queue entry. The truncation bit is set if the text area supplied by the requestor is smaller than the message text. The queue overflow bit is set if at least one message has been lost due to a queue-full condition.

This communicate is implicitly conditional; if no message is in the queue, a conditional fail (#0000) is returned in the fetch message. If the queue is not empty, the first message is taken off, the text is transferred and the header is appropriately initialized.

SECTION 6
MEMORY MANAGEMENT AND PROGRAM ENVIRONMENT

VIRTUAL MEMORY

DEFINITION

'Virtual memory' is used to mean the way the disk is made to behave as an overflow to the internal memory. This enables programs that require more total memory than that available on a particular host machine to execute on that machine, since those parts that are currently required for the execution are made present in the internal memory at any one time by the MCP.

STORAGE MEDIA

Under CMS there are three levels of storage: main memory, disk storage, and backing storage. In main memory, each byte has a unique address and can be accessed extremely rapidly. The maximum amount of main memory which can be connected to the processor is limited by its addressing capabilities and is organized in 64K byte pages for the B 90 processor. The processor can execute instructions and manipulate data only from main memory, although it is also able to transfer information between main memory and peripheral devices.

Disk storage is connected to the system as a peripheral device and is capable of storing from 1.2 million to 80 million bytes on one input-output channel. Details relevant to various disk types supported by CMS are presented in table 2-1. Backing storage normally takes the form of magnetic tape or cassette. A cassette is a compact, easy to use, method of storing information in large quantities. Each cassette can hold up to 293K bytes of information. However, the access time to a byte at the end of a tape can be measured in minutes.

SEGMENTATION

The processor can only execute instructions from main memory. Therefore, in order to execute any program, it must be brought or loaded into memory. However, it is not necessary to have the whole program in memory, since only parts of the program are required at any one time. For example, most programs can be logically divided into three parts; initialization, process and termination. They need not all be present at any one time. By suitably segmenting the program, and keeping segments on disk until they are required in main memory, space required to run a program can be minimized. The movement of segments to and from main memory is normally a system function, since in a multi programming environment, no single user program can optimize system resource utilization. This movement is transparent to the user programmer and indeed to the program which simply assumes the presence of segments in memory. The MCP loads these segments into memory if they are currently on disk. If no memory is available, it may be necessary to remove, from main memory, segments which are currently in use. The strategy followed in selecting such segments is described later under 'MEMORY MANAGEMENT'.

Figure 6-1 shows the history of a sample segmented program. At program initiation, code segment 1 performs an initialization using data segment 2. At B, code segment 1 calls code segment 2 which, at C begins using data segment 3. At D, code segment 2 calls a subroutine in code segment 3 which does not require data segment 2 but accesses data segment 3. At E, code segment 3 returns to code segment 2. F and G represent a second call on the subroutines in code segment 3. At H, code segment 2 finishes processing and hands over to code segment 4 which at 1 finishes with data segment 3 and at J the program terminates.

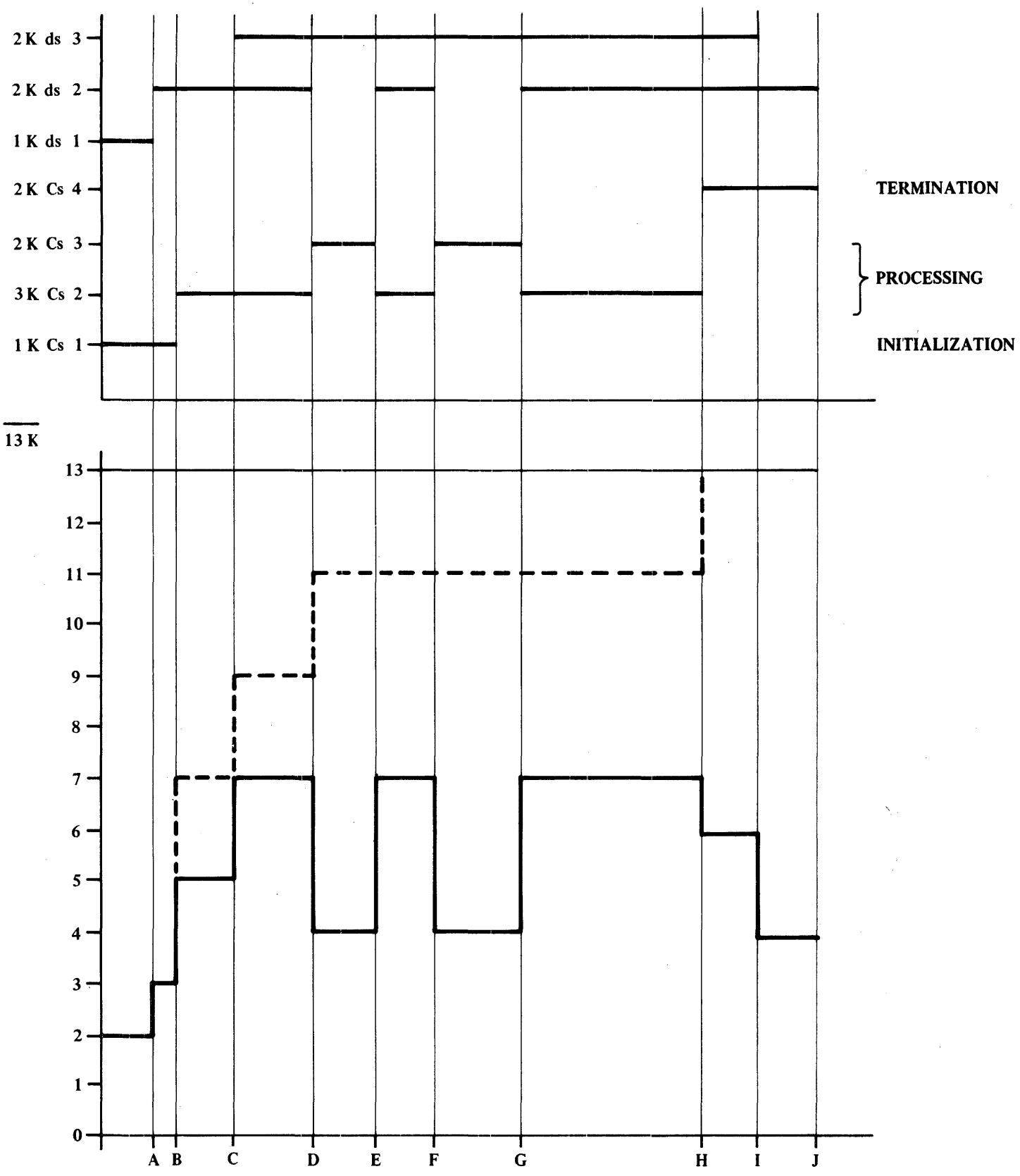


Figure 6-1. Sample Program Segmentation

In figure 6-1, A shows which segments are in use at any one time. Note that only one code segment is in use at any one time but that one code segment may access more than one data segment.

In figure 6-1, B shows three alternative memory management strategies. If no segment overlaying is possible, all segments must be in memory throughout program execution, and 13K bytes of memory are required. If segments are brought into memory as they are accessed, but never removed from memory, the dotted line history results where memory requirements gradually climb to 13K bytes at the end of job (that is, when all segments have been accessed). The third line shows a minimum efficient memory requirement where only those segments actually being accessed are kept in memory. In a virtual memory system, the actual memory requirements are in practice between the last two; segments are brought into memory as they are required but when no more room exists, segments are removed from memory to make room for the required segment.

Figure 6-2 shows an enlarged view of the time between points D and E in figure 6-1. Here we assume that there are only 6K bytes available to run the program. Code segment 2 is always in memory since it is executing, but data segments 2 and 3 are used alternately. Since all three segments cannot fit into the 6K limit, data segments 2 and 3 are alternately brought into memory and removed to disk.

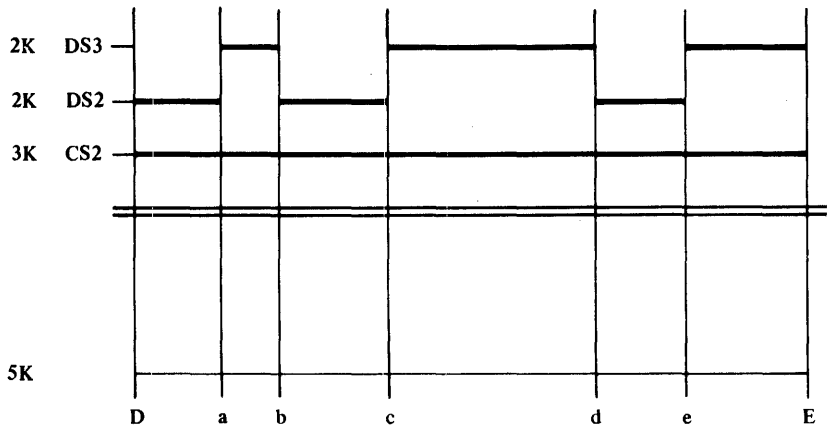


Figure 6-2. Memory Requirements

This swapping of segment takes time and the program runs correspondingly more slowly. Indeed it is possible to run the program with 3K bytes of memory available, in which case code and data segment take it in turn to be brought into memory at the instant that they are required. Note that the memory requirements quoted are for the object program only; the interpreter and run structure, used to maintain the information about segments, take up additional memory. In the case of running with only 3K bytes for the program, the overhead of swapping segments can take a large percentage of the total execution time and may in fact exceed the time spent performing useful functions for the user. This situation is called 'thrashing' and is described below.

Thrashing

Thrashing occurs when the group of segments which a program accesses, within the time taken for one segment to be recovered from disk (its working set), will not fit into the available physical memory. If a program has a working set which will not fit into the available physical memory, then in most cases when an absent segment is made present, the space for it is obtained by making absent another segment which can be accessed shortly after. This situation is detected by the MCP by counting the

number of segment address resolutions and the number of address resolutions which require access to a non-present segment. The ratio of address resolutions to absent segment accesses is calculated and if it shows a low value, thrashing is assumed to be occurring. When thrashing is detected, or no memory can be found, and more than one program is running, a program chosen as described below is swapped out by first making all its unallocated segments absent and then removing its locked slice to disk also. The program is no longer a candidate for immediate execution. If only one job is running it is not swapped out.

Figure 6-3 shows the approximate relationship between the memory available and the execution time of a program. With less memory than indicated at point A, the program cannot execute, since at least one segment is of size A. As more memory is made available, the execution time drops until at point B the overhead of segment overlaying becomes tolerable. By point C, there is sufficient memory available to hold the whole program and no segment swapping is required.

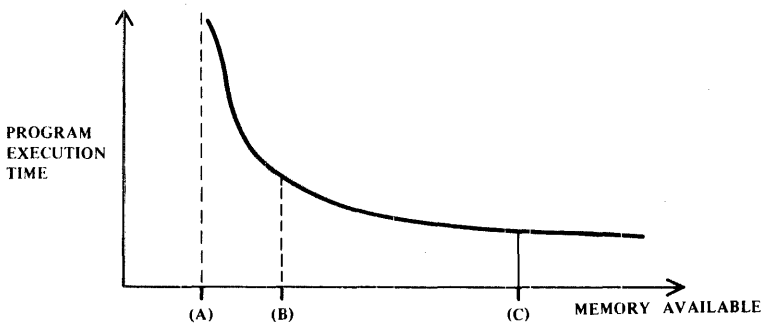


Figure 6-3. Effect of Memory on Throughput

PROGRAMS AND TASKS

A program is the smallest unit of code whose execution can be initiated independently, irrespective of the presence of other programs, except that all executing programs must share a finite pool of system resources. A task, or job, refers to a particular execution request. All concurrent incarnations of the same program use the same copy of the S-code, and all concurrent interpretations of the same language use the same copy of the language interpreter. The number of tasks (of which several can be incarnations of the same program) that can be concurrently executed (multi-programmed) on a machine is implementation dependent.

A user can interact with the system either programmatically through the ZIP communicate, or via the system SPO or ODT. In either case, the language used in such a communication is referred to as the System Control Language (SCL). For a detailed description of SCL, refer to [1]. The MCP completes the relevant requested action, and the originating task or operator is notified of success or failure before a further message is accepted from any other source. However, established unrelated tasks proceed asynchronously.

TASK PRIORITIES

Tasks are divided into three classes of priority in obtaining use of the processor for execution. All incarnations of the same program have the same class since such information is recorded in the object

file and can only be changed through a utility (MODIFY). The characteristics of these classes are exhibited in table 6-1.

Table 6-1. Classes and Relative Priorities

Relative Priority	Class	Organization of Priority within Class	Expected Nature of Tasks in Class
Low	A	Prefer tasks which do physical I/O	Regular work
Medium	B	Reverse historical (LIFO)	Utilities
High	C	Prefer tasks which do physical I/O	Sleepy real time transaction driven (data communications)

TASK STATES

There are three basic states in which a task can be during its lifetime; executing, delayed or suspended.

An executing task can either be ready to use the processor, or waiting for completion of physical I/O which can be a data communication message, virtual memory, or any other type of I/O.

Delayed tasks were previously executing but have been swapped out by the virtual memory routines because of their low priority when thrashing was detected. Delayed tasks are only changed to executing (and eventually swapped in) when another executing task either volunteers to be swapped out or runs to completion.

Suspended tasks were previously executing but have encountered circumstances requiring operator action (such as providing a missing input file). Tasks can also be directly suspended by the operator (by using the ST SCL command).

SWAPPING

This is also referred to as 'roll out to disk' and can be defined as a mechanism created to assist virtual memory routines in the choice of task to swap out (copy all its resident segments and its run structure out to disk.) Executing class C tasks can volunteer to be swapped out pending completion of a Data Communication receive, and all suspended tasks are deemed to have volunteered. When the virtual memory routine decides that there are too many tasks competing for physical memory, it swaps out the lowest priority task that has volunteered. If there are no volunteers available to be swapped out, the virtual memory routine changes the state of the lowest priority executing task to "delayed" and swaps it out. If a volunteer is swapped out, it is not "delayed", because when its requirements are satisfied it will be swapped in again.

MEMORY MANAGEMENT

The virtual memory subsystem maintains tables within the memory structure which enables any interpreter to find where any program's segments are located in memory. Also included in these tables is information indicating that status, length and disk address of the segment. For a detailed discussion of these tables, refer to the appropriate implementation dependent section. Whenever a program requires data, or code in a segment, the interpreter must first determine if the segment is present in memory, since main memory is the only place from which data can be directly accessed. If the segment is present, the base address of the segment can be determined from the virtual memory tables. Adding to this the displacement within the segment of the required data item gives the absolute location in main memory of the required data item. If the required segment is absent (that is, no copy exists in the main memory), then the interpreter must request the virtual memory subsystem to bring a copy of the segment into main memory. In fetching a segment from disk, the virtual memory subsystem must find an area in main memory which is not being used and is of sufficient size to store the required segment.

Figure 6-4 describes the algorithm used in finding space in memory to fetch a segment from disk. In step 1, only those areas of memory marked as unallocated are examined. If this search fails to find sufficient space, then a search is made for a contiguous area of memory, sufficient for the required segment, and containing only unallocated, read only, memory and up to one read/write segment (step two). In this search only segments not recently used are allowed, and all segments examined are marked as being not recently used.

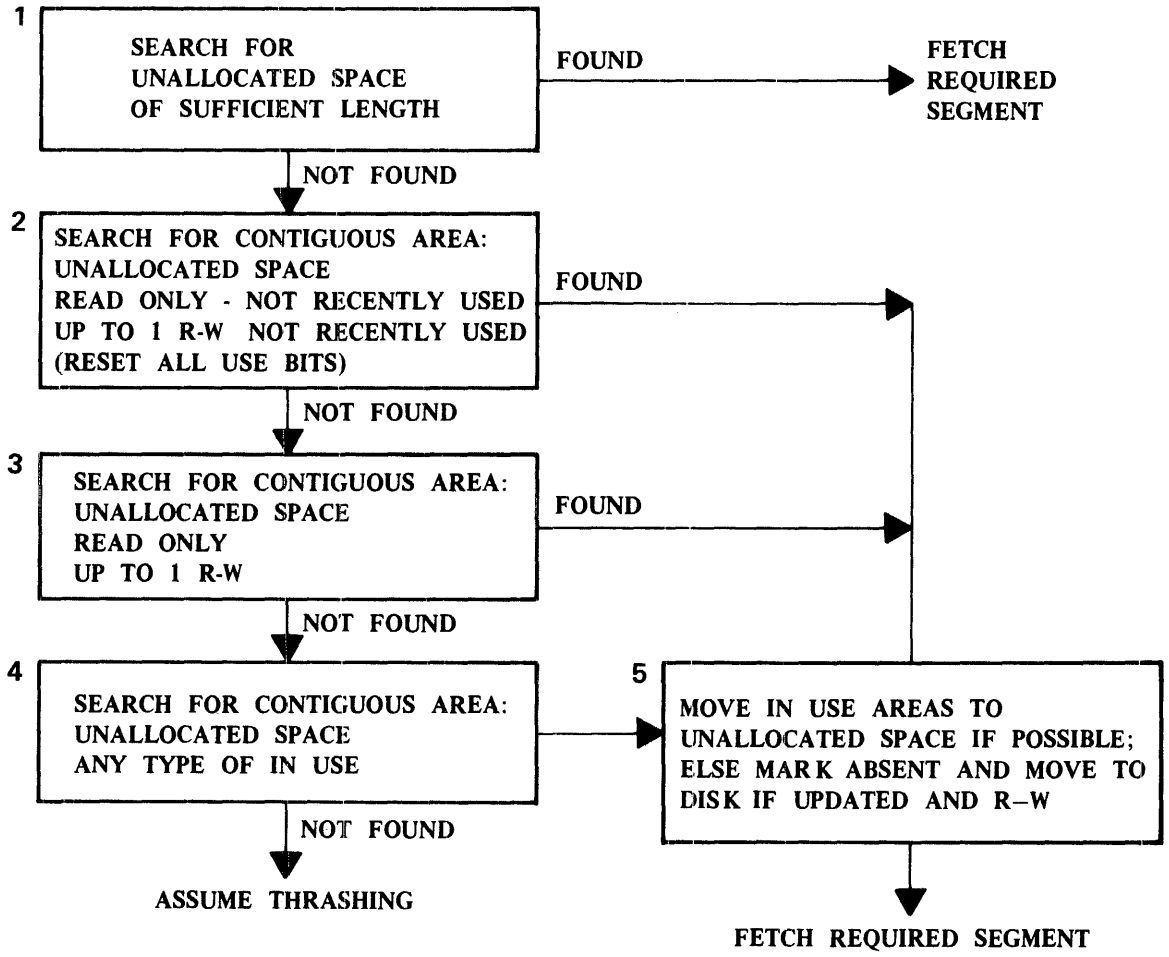


Figure 6-4. Virtual Memory Overlay Algorithm

Step three is the same as step two, though now more segments will be marked as not recently used and therefore this search may be successful where step two was not. If step three fails to find sufficient main memory, step four admits any type and number of segments in an attempt to find main memory space for the required segment. If step four fails, which can only happen when segments locked into memory take up too much room to allow the required segment into memory, then, in a multi-programming situation, one program may be temporarily suspended, freeing its allocated memory for other programs. If only one program is executing, the system is unable to carry on execution of the program.

If one of step 2 through 4 succeeds in finding a suitable area of memory, then there will be at least one part of the selected area which is assigned to another segment: such a segment is termed a tenant. In step five,

the selected area is cleared of all its tenants in preparation for moving the required segment from disk. For each tenant in the selected area, an alternative unallocated area is searched for, and if found, the tenant moved in.

Tenants which have been updated are moved first, then those tenants not yet altered in main memory. If this clearing process leaves tenants still in the selected area, then it is necessary to make them absent from memory. In the case of segments not yet updated in main memory, the copy on disk is the same as the copy in main memory and the segment is simply marked absent. If the segment has been updated, then it must be written to the disk copy before it is marked absent. When step five clears the selected area of all tenants, the required segment can be read into memory from the disk and the program requesting the segment can be restarted.

SECTION 7
CMS VIRTUAL MACHINES

INTRODUCTION

CMS is the implementation of a single virtual machine on a variety of host hardware designs. In this context a virtual machine is defined as the emulation, by software means alone, of a theoretical hardware design. The virtual machine is what the user 'sees' and uses, rather than the particular piece of hardware that hosts the virtual machine. The component parts of the CMS virtual machine are the source languages (COBOL, RPG, MPLII and NDL) and their respective compilers, which produce S-code object files, the language interpreters which execute the S-code and which interact with the MCP when necessary, and finally, the MCP itself which controls the total user environment and the interface with the operator.

Each high-level language along with its interpreter can be considered as a virtual machine in its own right operating within the framework of the CMS virtual machine since each language defines its own specific environment.

CMS supports four different languages; COBOL, RPG, MPLII and NDL. COBOL and RPG share the same S-code and interpreter. MPLII and NDL each have their own S-code and interpreter.

COBOL/RPG VIRTUAL MACHINE

The virtual machine has an 8-bit wide basic memory access. This byte can, in turn, be subdivided into two 4-bit digits. All items, excluding certain 4-bit data items, start on byte boundaries.

OPERAND TYPES

The virtual machine fetches, decodes and executes S-instructions. An S-instruction consists of an S-operator which is encoded into one byte giving a range between 0 and 255. Those codes which are not used, are recognized by the interpreter as invalid and cause the termination of the program. The S-operator byte is followed by arguments. The format and meaning attached to these arguments depends on the particular S-operator. However, in general, those arguments constitute the operators on which the S-operation is to operate. An operand can be one of three types, as distinguished by a descriptor consisting of one or more bytes. For the purpose of discussion, assume that the descriptor has the format shown in figure 7-1.

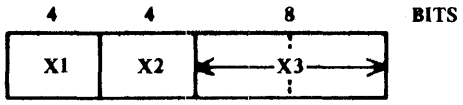


Figure 7-1. Data Descriptor

The three types an operand can have are as follows:

1. In-line literal.

This is distinguished by X1 = #F. X2 is subdivided into two fields which indicate the type and length of the literal to follow, as shown in figure 7-2. X3 in this case does not exist and the literal string follows X2.

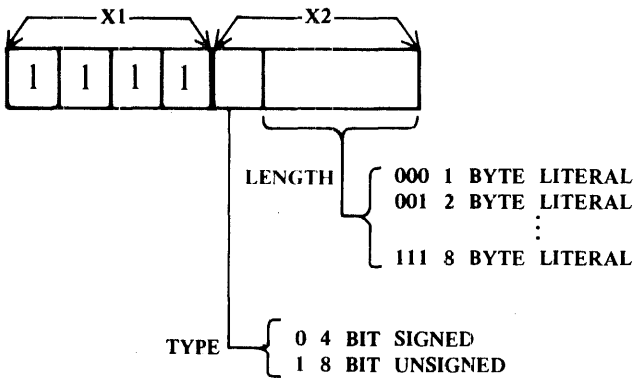


Figure 7-2. In-line Literal Descriptor Format

The literal string must be between one and eight bytes long. Longer literals are not presented in line but have an indirect reference as described in the remaining types. The format of the data within the literal string can be either 4-bit signed with a leading sign or 8-bit unsigned. Since the literal must be an integral number of bytes long, 4-bit literals must consist of an odd number of digits (due to the sign which is #5 for negative and any other value for positive. The S-machine will insert #3 if the result of a calculation is positive. A 4-bit literal with an even number of digits requires a padding with a leading zero digit so as to make up the length to an integral number of bytes.

2. Current Operand Table Index

This is an indirect data reference (COPX) made via a table called the current operand table (COP TABLE) which contains the attributes and address of the data item. There are two variations depending on whether the index is greater than 223 (LONG COPX) or not (SHORT COPX). These variations are distinguished as follows:

> $0 \leq X1 \leq$ #D: X3 not present: short COPX: this allows the top 224 entries of the table to be referenced.
>
> X1 = #E, $0 \leq X2 \leq 14$: long COPX: this allows a further 3840 entries to be referenced, the index being X2 x X3 + 224.

A short or long COPX argument is an index to an entry in the current operand table (COP table). Each entry in the COP table is a multiple of four bytes in length. The value of the COPX is the number of four byte units by which the first bit of the associated descriptor is displaced from the base of the table. If a COPX refers to an operand which is subscripted or indexed, then the reference to the subscripts or indexes follows immediately after the primary COPX. Each of the secondary refer-

ences can be a COPX or a literal (such a choice will be denoted by OPND) but cannot itself be subscripted or indexed. If a subscripted or indexed secondary reference is encountered during execution, the program is terminated.

3. In-line COPX.

This argument has X1 = #E and X2 = #F (X3 is not present) immediately followed by an in-line descriptor which is identical to the format of a COP table entry, except that the extension is not padded out to be a multiple of four bytes. If an in-line COPX is referencing a subscripted or indexed operand, then the rules outlined above hold.

COP TABLE FORMAT

The COP table consists of a set of entries of which each descriptor contains the attributes of one data item, or of an array of identical data items. A base descriptor contains the attributes of a data item which is neither subscripted nor indexed, or of any individual member of an array or table. It can be either four bytes or eight bytes in length. The entry for an array or table is a base descriptor followed by an extension which describes the subscripting or indexing necessary to identify a member of the array. The extension consists of a number of two-byte fields. Each base descriptor starts at a multiple of four bytes from the base of the COP table. If the overall length of an entry, base plus an extension, is not a multiple of four bytes, two bytes are left unused before the next entry.

The base descriptor has two forms; a short form which is four bytes long, and a long form which is eight bytes long. The long form has the same fields as the short form, but the displacement and length fields are extended to sixteen bits. The short and long forms of the base descriptors are described in figures 7-3 and 7-4 respectively.

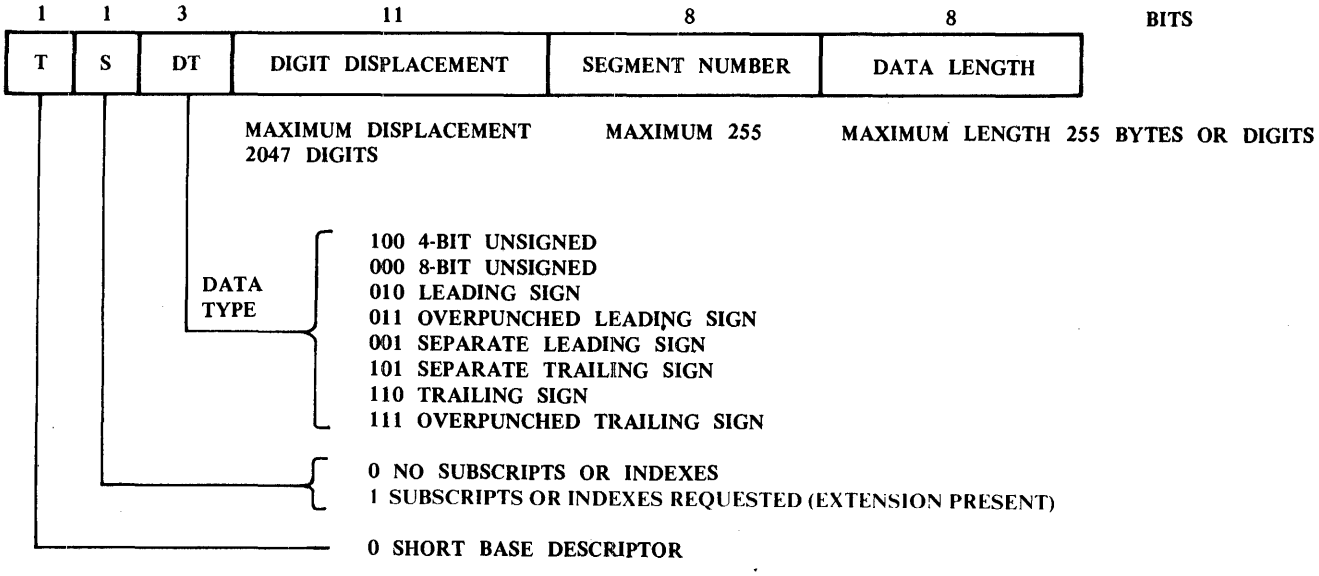


Figure 7-3. COP Table Base Entry – Short

A long base is used if either of the displacements or data length values require a longer field than is allowed for it in the short base. If the displacement of an 8 bit field, after modification by subscripting or indexing is odd, the last significant bit is discarded. Therefore, the compiler should provide the necessary digit filler when this happens to be the case.

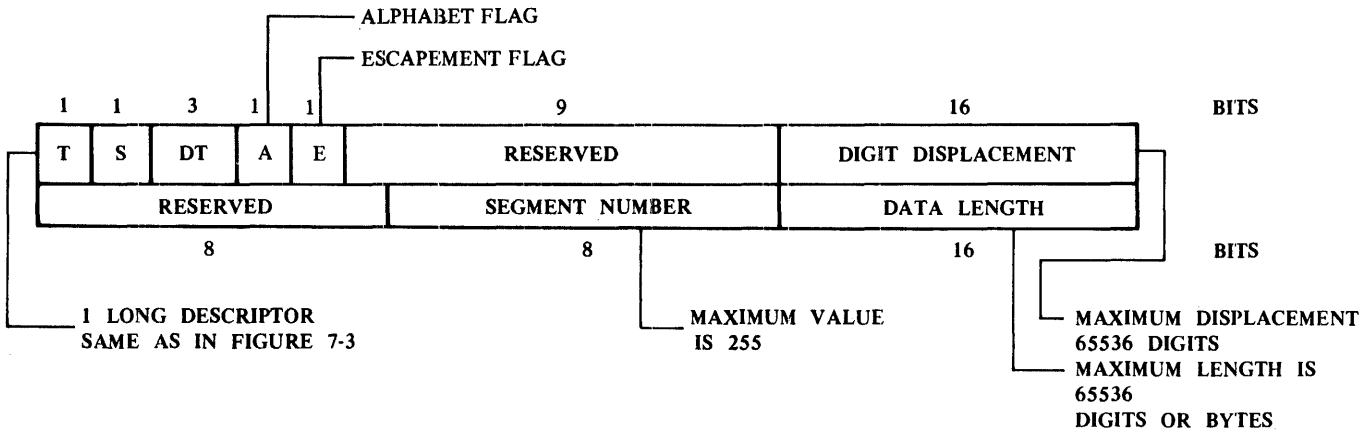


Figure 7-4. COP Table Base Entry – Long

The data length field contains, in binary, the length of the data item. This is the number of data units (digits or characters) in the field. It does not include the space occupied by a sign in a four bit or a separate sign in an eight bit field. The binary value in each case is one less than the length of the field. A four bit field cannot be longer than 15 digits and a sign.

The base for an operand which is subscripted or indexed is followed by an extension which consists of a sequence of two byte fields. For descriptors in the COP table, if there are an odd number of fields the last is followed by two unused bytes, so that the next entry will start at a multiple of four bytes from the base of the COP table. The two unused bytes do not appear after in-line descriptors.

Each two byte field contains a one-bit flag followed by 15 data bits. The format of the extension is illustrated in figure 7-5.

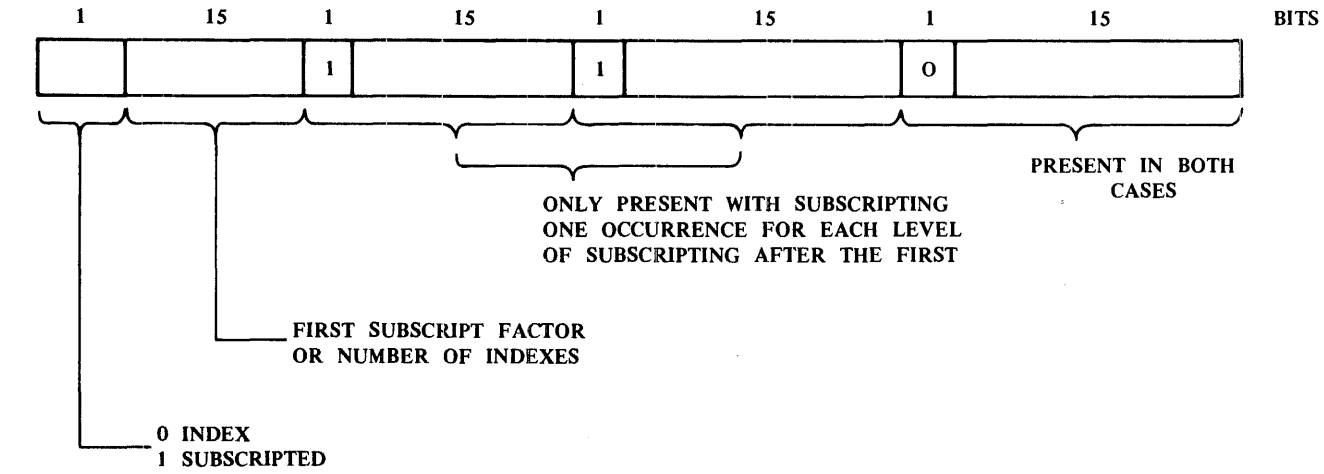


Figure 7-5. Extension Format

If indexing is specified, the second extension field is the table bound field. If subscripting is specified, the first extension is the first subscript factor and there is one subscript factor for each level of subscript required. The subscript factors are allowed by the table bound field. The table bound field terminates the extension field. A data item cannot be both subscripted and indexed.

The flag of a Subscript Factor is set to 1. The fifteen data bits contain the binary factor by which the corresponding subscript value is to be multiplied to obtain the digit address required. The factor is the digit displacement between successive elements of the table. The effect of an odd subscript factor associated with an 8 bit data item is as follows: the odd factor is used in the calculation and the potentially odd bit is discarded when the final displacement has been calculated. There is one subscript factor for each level of subscripting required. The end of the sequence of subscript factors is indicated by the occurrence of the table bound, which has its flag bit set to 0. Although this method puts no limit on the number of subscripts, the COBOL language permits a maximum of three levels of subscripting. The data references to the subscripts follow the primary COPX in the S-instruction. Each reference is an OPND which may be in-line or a literal but must not be subscripted or indexed. If a subscripted or indexed subscript is detected during execution, the program is terminated.

The subscript value is reduced by one prior to multiplication by the factor. If the result of this subtraction is negative, the program is terminated. The sum of the products is formed and added to the displacement specified in the base descriptor to give the effective digit displacement in the table. If the sum of the products exceeds the table bound or if an overflow is detected at any time during the computation, the program is terminated.

The flag bit of an index count is set to 0. The fifteen data bits are divided into two sub-fields. The most significant seven bits are reserved for future use. The least significant eight bits (the second byte) contain a binary number which is one less than the number of levels of indexing required. The corresponding number of data references to index variables follow the primary data reference in the S-introduction.

Although a maximum of 256 levels of indexing can be specified, the COBOL language permits no more than three. The data reference to an index variable is an OPND, which may be in-line or a literal but must not be indexed or subscripted. If a subscripted or indexed index is detected during execution, the program is terminated. There is no restriction on the data type of the index variables but the value must lie between 0 and + 65535 inclusive, unless it is a literal. Literal values can be negative. The sum of the indexes is formed and added to the displacement specified in the base descriptor to give the effective digit displacement in the table. If this is odd for an 8 bit item, the odd bit is discarded (round down). If an overflow is detected at any time during the computation, or if the sum of the indexes exceeds the table bound at the end of the computation, the program is terminated. The extension field after the index count is assumed to be the table bound.

The flag bit of the table bound is set to 0. The 15 data bits contain a binary value which is the highest digit displacement which can be generated by indexing or subscripting calculation. If this value is exceeded, an error is recognized. This table bound indicates the end of a sequence of subscript factors. A table bound occurs once in each descriptor extension and is the last item in that extension.

SIGN REPRESENTATION

If a sign is specified for four bit data items, it occupies one digit position which can be either trailing or leading. For 8 bit data items, the sign can be overpunched over the zone digit of either the leading or trailing byte. The values used are #3 for positive and #5 for negative. Alternatively, the sign can occupy a separate character (byte) which again can be leading or trailing. In this case, the values used are '+' for positive and '−' for negative.

Table 7-1. COBOL S-Instructions

SECOND DIGIT	FIRST DIGIT 0	1	2	3	4	5
0		BIS	ALIS	FCR	ILCR	TCR
1	INC1	BIL	ALIL	CDB	RLCR	MULS
2	INC	BES	ALES	CBD	FLASH	DIVS
3	ADD	BEL	ALEL	MVA	BITON	MVSGN
4	DEC1	BOIS	NTRIS	MVS	BITOF	PERLIS
5	DEC	BOIL	NTRIL	MVN	TESTB	PERLIL
6	SUB	BOES	NTRES	MVZ	SETON	PERLES
7	MUL	BOEL	NTREL	SMVN	SETOF	PERLEL
8	DIVQ	BNIS	XIT	MVT	BOFIS	PXITL
9	DIVQR	BNIL	CPA	CAT	BOFIL	SUSP
A	PXIT	BNES	CPS	EDT	BOFES	
B	GOPAR	BNEL	CPN	EDTE	BOFEL	
C	GTDI	PERIS	CPZ	AEDT	BONIS	
D	GTDE	PERIL	CRPT	AEDTE	BONIL	
E	OFV	PERES	CMPC	EXAM	BONES	
F	COFV	PEREL	COMM	SLCR	BONEL	

Table 7-1 shows the COBOL S-instructions, mnemonics, and hexadecimal code.

The arguments in the instruction definitions that follow are mostly expressed in terms of the following abbreviations:

COPX:	Current Operand index
OPND:	A data reference which may be an in-line literal
BASIC:	A one byte binary value specifying an intra-segment, short branch address
BADIS:	A two byte binary value specifying an intra-segment, short branch address
BADIL:	A two byte binary value specifying an intra-segment, long branch address
BADES:	A two byte binary value specifying an extra-segment, short branch address. The first byte denotes a code segment number and the second denotes an offset from the base of that segment
BADEL:	A three byte binary value specifying an extra-segment, long branch address. The first byte denotes a code segment number and the last two bytes denote an offset from the base of that segment
DADDR:	A two byte binary value specifying an offset from the base of data segment zero

The functions of the individual instructions are described in the following paragraphs. A data reference within squared brackets is used to denote the content of the data item referenced.

ARITHMETIC OPERATIONS AND OPERANDS

Any operand in an arithmetic operation can be either 4-bit or 8-bit format and either signed or unsigned. An unsigned source field is considered positive. If a result field is unsigned, the absolute value

of the result is stored.

The effect of a digit code greater than 9 in a source field is undefined.

If a source field is specified to contain more than 15 digits, only the least significant 15 digits will be used in the operation. If a result field is specified to contain more than 15 digits, the least significant 15 digits of the result will be stored right-justified and the field filled with zeros. The specified length of the field will be used in determining the location of the sign.

If a result field is not long enough to contain a result, the overflow is set and the result truncated at the most significant end. If a result field is longer than a result, the result is extended with leading zeros.

The source fields of an operation can overlap in any way. A result field can coincide with a source field. The result is undefined if there is a partial overlap of a source field and a result field, or if result fields overlap in any way.

All operands are considered to be decimal integers.

Increment

Format: INC OPND1, COPX2
Function: [COPX2]←[COPX2] + [OPND1]

Add

Format: ADD OPND1, OPND2, COPX3
Function: [COPX3]←[OPND1] + [OPND2]

Decrement

Format: DEC OPND1, COPX2
Function: [COPX2]←[COPX2] – [OPND1]

Subtract

Format: SUB OPND1, OPND2, COPX3
Function: [COPX3]←[OPND2] – [OPND1]

Increment By One

Format: INC1 COPX1
Function: [COPX1]←[COPX1] + 1

Decrement By One

Format: DEC1 COPX1
Function: [COPX1]←[COPX1] – 1

Multiply

Format: MUL OPND1 OPND2 COPX3
Function: [COPX3]←[OPND1]*[OPND2]

NOTE

The number of significant digits in the product is the same as, or one less than, the sum of the numbers of significant digits in the multiplier and multiplicand.

Divide Giving Quotient

Format: DIVQ OPND1 OPND2 COPX3
Function: [COPX3]←[OPND2] ÷ [OPND1]

NOTE

The number of significant digits in the quotient is the same as, or exceeds by one, the amount by which the number of significant digits in the dividend exceed the number of significant digits in the divisor. If the divisor has more significant digits than the dividend the quotient is zero. If the divisor is zero, the overflow flag is set and the contents of the fields are not changed.

Divide Giving Quotient and Remainder

Format: DIVQR OPND1 OPND2 COPX3 COPX4
Function: [COPX3]←[OPND2] ÷ [OPND1];
[COPX4]←[OPND2] – [COPX3]*[OPND1]

NOTE

The number of significant digits in the quotient is the same as, or exceeds by one, the amount by which the number of significant digits in the dividend exceed the number of significant digits in the divisor. If the divisor has more significant digits than the dividend, the quotient is zero. If the divisor is zero, the overflow flag is set and the contents of the fields are not changed.

The sign of the remainder is the same as the sign of the original dividend.

Scaled Multiplication

MULS, FRSCL1, OPND2, OPND3, COPX4

Form the product of the contents of the fields denoted by OPND2 and OPND3. Multiply this product by a power of ten specified by FRSCL1. Round if specified by FRSCL1 and store the result in the field denoted by COPX4.

FRSCL1 is a one-byte field consisting of three subfields, F, R and SCL.

F is bit 0 and is reserved for future use. It is ignored, but for compatible growth should be zero. R is bit 1. If R is zero, a truncated result is stored. If R is 1, a rounded result is stored.

SCL consists of the least significant six bits and is interpreted as a positive binary integer.

The result of the operation is defined by the following procedure:

1. Find the magnitudes of the values represented by OPND2 and OPND3.
2. Subtract 23 from SCL, giving S.
3. If S is greater than 30, set the result value to 0 and go to step 16.
4. If S is greater than −15, go to step 7.
5. If neither of the input values is zero, set overflow.

6. Set the result to 0 and go to step 16.
7. Form the product of the two input values. This will contain not more than 30 significant digits. If it contains fewer than 30 significant digits, then expand it to 30 digits by the concatenation of leading zero digits.
8. If S is greater than zero go to step 11.
9. Extend the product of the least significant end by the concatenation of $-S$ zero digits.
10. Take the 15 digits from the least significant end of the extended product as the result value. Go to step 15.
11. Discard S-1 digits from the least significant end of the product.
12. If fewer than 16 digits remain in the product, expand it to 16 digits by concatenation of leading zeros.
13. If R is 1, increase the value of the truncated product by 5.
14. Discard the least significant digit of the truncated product and take the next 15 digits from the least significant end of the product as the result value.
15. If any digit to the left of those taken to form the result value is non-zero, set overflow.
16. If the result value is zero, set the sign of the result positive, otherwise set the sign of the result according to the normal rules of arithmetic.
17. Store the result sign and value in the field denoted by COPX4 according to the rules for the storage of numeric values.

Scaled Division

DIVS, FRSCL1, FRSCL2, OPND3, OPND4, COPX5, COPX6

Divide a dividend, the contents of the field denoted by OPND4, by a divisor, the contents of the field denoted by OPND3. Store the quotient in the field denoted by COPX5 and the remainder in the field denoted by COPX6. FRSCL1 is a one-byte field which specifies the scaling and rounding to be applied to the quotient and whether or not the remainder is to be stored. FRSCL2 is a one-byte field which specifies the scaling and rounding to be applied to the remainder.

Each FRSCL is a one-byte field consisting of three subfields, F, R and SCL.

F is bit 0. F1 is 1 if a remainder is to be stored and zero otherwise. If F1 is 0, FRSCL2 and COPX6 are not present. If F2 is present, it is ignored.

R is bit 1. If R is 0, a truncated result is stored. If R is 1, a rounded result is stored.

SCL is the six least significant bits and is interpreted as a positive binary integer.

The result of the operation is defined by the following procedure:

1. Find the magnitude of the divisor, the contents of the field denoted by OPND3. If it is zero, set overflow and terminate the operation, leaving the contents of the quotient and remainder fields unchanged.
2. Find the magnitude of the dividend, the contents of the field denoted by OPND4.
3. Expand both dividend and divisor to a length of 15 digits by the concatenation of leading zeros.
4. Find L5, the length of the quotient field denoted by COPX5. If L5 is greater than 15, use 15 instead of L5 in all subsequent steps.
5. Set $S1 = L5 + SCL1 - 4$.
6. If S1 is less than or equal to zero, go to step 12.
7. Compute the first S1 digits of the quotient by repeated subtraction of the divisor from the dividend. The first S1 digits include any leading, non-significant zeros. Record the reduced di-

vidend as the remainder. Append one leading zero to the quotient, increasing its length to T1 = S1 + 1 digits.

8. If R1 is 1, compute one more digit of the quotient, add 5 to the quotient and then reduce the length of the quotient by one place by discarding the least significant digit.
9. If T1 is greater than 15, go to step 11.
10. Increase the length of the quotient to 15 digits by concatenating zero or more leading zeros. Go to step 13.
11. Remove T1-15 digits from the most significant end of the quotient. If any of the digits removed is non-zero, set overflow. Go to step 13.
12. Set the quotient value to zero and record the dividend value as the remainder.
13. If the quotient value is zero, or if the signs of the divisor and dividend are the same, set the sign of the quotient positive; otherwise set the sign of the quotient negative.
14. Store the quotient sign and value in the field denoted by COPX5 according to the rules for the storage of numeric values.
15. If F1 is zero, terminate the operation.
16. If the remainder value is zero, go to step 25.
17. Subtract 32 from SCL2 giving S2. If $S2 < 0$, go to step 20.
18. Extend the remainder value by the concatenation of S2 zero digits at the least significant end.
19. Reduce the length of the remainder value to 15 digits by discarding digits from the most significant end. If any discarded digit is non-zero, then set overflow. Go to step 25.
20. If $S2 \leq 15$, set the remainder value to 0 and go to step 25.
21. Extend the remainder value by the concatenation of $-S2$ zero digits to the most significant end.
22. Reduce the length of the remainder values to sixteen digits by discarding digits from the least significant end.
23. If R2 is 1, increase the magnitude of the remainder value by 5.
24. Reduce the length of the remainder value to 15 digits by discarding the least significant digit.
25. If the remainder value is zero, set the remainder sign positive; otherwise set the remainder sign to the sign of the dividend.
26. Store the remainder sign and value in the field denoted by COPX6 according to the rules for the storage of numeric values.

DATA MOVEMENT OPERATIONS

In general the fields involved in data movement operations can be in 4-bit formats signed or unsigned. Any restrictions on field type for a particular operation are specified in the description of that operation.

Unsigned fields are considered positive when appropriate.

For numeric move operations, data is placed right justified in the destination field. If the length of the data item differs from the length of the destination field, the data is truncated or filled with zeros at the most significant end. If the destination field is signed, the sign of the data is placed in the sign position regardless of any truncation of filling.

For alphanumeric move operations, data is placed left justified in the destination field. If the length of the data item differs from the length of the destination field, the data is truncated, or filled with spaces at the least significant end. If the destination field is of a separate signed type the sign byte is not changed.

The effect of total or partial overlap of a result field with any other field is not defined unless overlapped fields are explicitly permitted in the descriptions of individual instructions.

Move Alphanumeric

Format: MVA OPND1 COPX2
Function: [COPX2]←[OPND1]

If the data type of the origin field is 4-bit, the sign, if present, is ignored and the character representation of each of the other digits is moved to the appropriate byte in the destination field.

If the length of the destination field is greater than the length of the origin field, the destination field is filled on the right with spaces.

If the length of the destination field is less than the length of the origin field, the source data is truncated on the right.

8-bit source and destination fields may coincide. The result of any other overlap is not defined.

Move Spaces

Format: MVS COPX1
Function: [COPX1]←SPACES

NOTE

If data type of destination is not unsigned 8-bit, the program terminates.

Move Numeric

Format: MVN OPND1 COPX2
Function: Move 4-bit or 8-bit units from the origin field denoted by OPND1 to the destination field denoted by COPX2.

If the destination field is signed it receives the sign of the origin field, if that is signed, or a positive sign if the origin field is unsigned. If the destination field is unsigned, the sign, if any, of the origin field is ignored.

If the data type of the destination field is 8-bit the character representation of each digit, or the corresponding signed digit representation, is moved to the appropriate byte in the destination field.

Move Zeros

Format: MVZ COPX1
Function: [COPX1]←Zeros of appropriate type

NOTE

If destination is signed, it is given a positive sign of appropriate type.

Scale Move Numeric

Format: SMVN OPND1, V, SCL, COPA2
Function: If V = 0, $[COPX2] \leftarrow [OPND1] \times 10^{SCL+1}$
If V = 1, $[COPX2] \leftarrow [OPND1] \div 10^{SCL+1}$

NOTE

Rules of MVN also apply here. V and SCL occupy one byte, V being the most significant bit.

The sign is preserved and moved to the appropriate position in the destination field if that is signed.

Move Translate

Format: MVT OPND1, OPND2, COPX3

Function

Move a translated copy of a source field, denoted by OPND1, into a destination field denoted by COPX3. The translation is specified by a 256-byte table denoted by OPND2. The data types of all three fields are ignored and they are assumed to be unsigned 8-bit.

Translation is character by character from left to right. Each character from the source field is used as an 8-bit binary value to index into the translate table.

The character which is located is copied from the translate table to the destination field. The length of the translation table may be less than 256 bytes, but if any source character has a binary value greater than the length of the table, the program is terminated.

If the length of the source is less than the length of the destination field, the destination field is filled on the right with spaces.

Concatenate

Format: CAT M, COPX0, OPND1 ..... OPNDN
Function:

Move a string consisting of the N fields denoted by OPND1 to OPNDN in the order specified into a destination field denoted by COPX0 according to the rules for Move Alphanumeric. M is a one byte field which contains a binary value, M, one less than the number of source fields, N, to be concatenated. If M is zero, the operation has the same effect as MOVE ALPHANUMERIC. If M is 255, then 256 source fields are concatenated.

If the destination length is greater than the combined source lengths, the destination field is filled on the right with spaces.

If the destination length is less than the source length, the concatenated string is truncated on the right.

Any of the source fields may overlap, but if any of them overlap the destination field, the result of the operation is not defined.

Table 7-2. Edit Micro-Operators

OPERATOR	MNEMONIC	OPERATION
0000R	MVD	Move digits
0001R	MVC	Move characters
0010R	MVS	Move suppress
0011R	FIL	Fill suppress
0100N	SRD	Skip reverse destination
0101T	INU	Insert unconditionally
0110T	INM	Insert on minus
0111T	INS	Insert suppress

(continued)

Table 7-2. Edit Micro-Operators

OPERATOR	MNEMONIC	OPERATION
1000T	INF	Insert float
1001T	EFM	End float mode
1010I	ENZ	End non-zero
1011I	EOM	End of mask
1100I	SZS	Set Z = 1 start zero suppress
1101I	CCP	P:Complement check protect
OTHERS	–	Error: Program terminates

Edit Operations and Edit Micro Operators

The valid micro-operators used are listed, with their corresponding mnemonic and operation in table 7-2. The second digit of the operator as shown in the table can be R, N, T, or I, which stand for the following:

'R' indicates a 4-bit binary value when used as a repeat count 0000 representing no repeat: the operation is done once only in this case.

'N' indicates a 4-bit binary value used to skip over a number of destination 8-bit units. The value 0000 represents no skip.

'T' indicates a 4-bit binary value which is used to:

1. Index into a table of editing constants (see table 7-3).
2. Indicate a conditional selection between two table constants. Or
3. Indicate an editing constant in-line with the edit-operator string. The next edit operator follows the constant.

Table 7-3 indicates the normal edit table constants as well as the conditional and unconditional selection of constants associated with the value 'T'.

'I' indicates a 4-bit field which is not used. Its contents are ignored.

Table 7-3. Edit Table Constants

T	Table Entry	Mnemonic	Unconditional or Conditional Constant
0000	'+'	PLU	
0001	'–'	MIN	
0010	' '	BLK	
0011	'*'	AST	
0100	'.'	DPT	
0101	','	CMA	
0110	'$'	CUR	
0111	'0'	ZRO	
1000		SPM	Either entry 0 or 1
1001		SBM	Either entry 2 or 1
1010-1111		LIT	In line 1 byte constant

Associated with the edit instruction, there are three switches, known as S for sign, Z for zero suppress and P for check protect. Initially Z and P are set to zero. They are set and reset as specified in the description of the individual micro-operators. S is set to 1 if the source field has a negative sign and to zero otherwise. Unsigned fields are considered positive.

The occurrence of any combination of switches and parameters not described for any micro-operators causes the program to terminate.

For Alphanumeric edits only three micro-operators are recognized: MVC, INU and EOM. The operation is terminated by EOM and not by the exhaustion of source or destination fields.

In the following paragraphs the individual edit micro-operators are discussed.

Move Digit: Set Z to 1, ending the zero suppress state. Move an appropriate unit (4-bit digit or 8-bit character) from the source field to the destination field. If a 4-bit unit is moved, append the four bits 1111 to the left before storing in the destination. If an #8-bit unit is moved, substitute the four bits 1111 for the left-most four bits of the 8-bit unit.

Move Character: Set Z to 1, ending the zero suppress state. Move an appropriate unit from the source field to the destination field. If a 4-bit unit is moved append the four bits 1111 to the left before storing it in the destination. If an 8-bit unit is moved it is moved unchanged.

Move Suppress: Source $\neq$ 0 Move Digit

Z = 1 Source = 0 Move Digit
Z = 0 P = 0 Source = 0 Move table entry 2 (blank)
Z = 0 P = 1 Source = 0 Move table entry 3 (asterisk)
Fill Suppress: P = 0 Move table entry 2 (blank)
P = 1 Move table entry 3 (asterisk)
Skip Reverse Destination: Adjust the address pointer of the destination field to skip backwards (lower address) N bytes
Insert Unconditionally: Move table entry T as indicated below to the destination field:

T = 0....7, Move table entry T
T = 8, S = 0, Move table entry 0 (Plus)
T = 8, S = 1, Move table entry 1 (Minus)
T = 9, S = 0, Move table entry 2 (Blank)
T = 9, S = 1, Move table entry 1 (Minus)
T>9 Move in-line character

Insert on minus: move table entry T as indicated below to the destination field:

S = 0 P = 0 Move table entry 2 (Blank)
S = 0 P = 1 Move table entry 3 (Asterisk)
S = 1 T = 0, 1....7 Move table entry
S = 1 T = 8 Move table entry 1 (Minus)
S = 1 T = 9 Move table entry 1 (Minus)
S = 1 T > 9 Move one in-line character
S = 0 T > 9 Skip over the in-line character

Insert Suppress: Move the table entry T as indicated below to the destination field:

Z = 0 P = 0 Move table entry 2 (Blank)

Z = 0 P = 1 Move table entry 3 (Asterisk)
Z = 1 T = 0....7 Move table entry T
Z = 1 S = 0 T = 8 Move table entry 0 (Plus)
Z = 1 S = 1 T = 8 Move table entry 1 (Minus)
Z = 1 S = 0 T = 9 Move table entry 2 (Blank)
Z = 1 S = 1 T = 9 Move table entry 1 (Minus)
Z = 1 S = 1 T > 9 Move in-line character
Z = 1 S = 0 T > 9 Move table entry 2 (Blank) and skip over the in-line character

Insert float: Move the table entry T and/or perform the micro-operation move digit as indicated below:

Z = 1 Move digit
Z = 0 source = 0 P = 0 (Blank) Move table entry 2
Z = 0 source = 0 P = 1 (Asterisk) Move table entry 3
Z = 0 source ≠0 T = 0....7 Move table 7, then move digit
Z = 0 source ≠0 T = 8 S = 0 Move table entry 0 (Plus) then move digit
Z = 0 source ≠0 T = 8 S = 1 Move table entry 1 (Minus) then move digit
Z = 0 source ≠0 T = 9 S = 0 Move table entry 2
Z = 0 source ≠0 T = 9 S = 0 Move table entry 2 (Blank) then move digit
Z = 0 source ≠0 T = 9 S = 1 Move table entry 1 (Minus), then move digit
Z = 0 source ≠0 T > 9 Move in-line character, then move digit

If T>9 and Z = 1 or Z = 0 and source = 0, skip over in-line character
End Float Mode: Move the table entry T as indicated below to the destination field:

Z = 0 T = 0,....7 Move table entry T
Z = 0 S = 0 T = 8 Move table entry O (Plus)
Z = 0 S = 1 T = 8 Move table entry 1 (Minus)
Z = 0 S = 0 T = 9 Move table entry 2 (Blank)
Z = 0 S = 1 T = 9 Move table entry 1 (Minus)
Z = 0 T > 9 Move in-line character
Z = 0 T > 10 No operation
Z = 1 T > 9 Skip over in-line character

End Non-Zero: Terminate the edit operation if any non-zero source character or digit has been moved, otherwise continue the next in-line operator.
End of Mask: Terminate the edit operation.
Start Zero Suppress: Set Z to the zero state.
Complement Check Protect: Complement the state of P.
The edit S-operations are now described.

Edit

Format: EDT OPND1, COPX2, DADDR3
Function: Move data from the source location denoted by OPND1 to the destination location denoted by COPX2 under the control of the micro-operator string contained at the location denoted by DADDR3. DADDR3 is a two byte field which contains in binary the displacement of the micro-operator string relative to the base of data segment zero.

Edit with Explicit Mask

Format: EDTE OPND1, COPX2, MASK3

Function: Move data from the source location denoted by OPND1 to the destination location denoted by COPX2 under the control of the micro-operator string denoted by MASK3. The format of the mask is a single byte length field followed by the micro-operator string. The length field contains a binary number which is one less than the length in bytes of the micro-operator string, not including the length field itself.

If the micro-operator string is not correctly terminated (by the EOM code) the execution of the program is terminated.

Alphanumeric Edit

Format: AEDT OPND1 COPX2 MASK3
Function: Move data from the source location denoted by OPND1 to the destination location denoted by COPX2 under the control of the micro-operator string contained at the location denoted by DADDR3. DADDR3 is a two byte field which contains, in binary, the displacement of the micro-operator string relative to the base of data segment zero.

Alphanumeric Edit with Explicit Mask

Format: AEDTE OPND1 COPX2 MASK3
Function: Move data from the source location denoted by OPND1 to the destination location denoted by COPX2 under the control of the micro-operator string denoted by MASK3. The format of the mask is a single-byte length-field followed by the micro-operator string. The length-field contains a binary number which is one less than the length in bytes of the micro-operator string, not including the length itself. If the micro-operator string is not correctly terminated (by the EOM code) the execution of the program is terminated.

Examine

Format: EXAM T, OPND1, OPND2, OPND3, OPND4, COPX5
Function: Examine the field denoted by OPND1, tallying and/or replacing a variable number of 8-bit characters according to the contents of T. The 8-bit character to be tallied or replaced is contained in the field denoted by OPND3. The 8-bit character to be used as a replacement is contained in the field denoted by OPND4. The tally is stored in the field denoted by COPX5. Tallying and replacing can be specified to start before or after the first occurrence of a character, specified by the contents of the field denoted by OPND2.

The field denoted by OPND1 must be of 8-bit type, signed or unsigned. If it is signed, the sign is ignored. Each of the fields denoted by OPND2, OPND3 and OPND4 must be of 8-bit, unsigned type. If any other type is specified, the execution of the program is terminated.

OPND2 is only present if the 'before' or the 'after' option is set in T. The field denoted by OPND2 must be one character in length, or the execution of the program is terminated. The first occurrence in the field denoted by OPND1 of the character specified by OPND2 defines the first or last character position not to be examined, according to the option set in T.

OPND3 is not present if the 'characters' option is set in T. The field denoted by OPND3 must be one character in length, or the execution of the program will be terminated. Occurrences in the field denoted by OPND1 of the character specified by OPND3 are tallied or replaced, according to the option set in T.

OPND4 is present only if the 'replacing' option is set in T. The field denoted by OPND4 must be one character in length, or the execution of the program will be terminated. Each occurrence in the

field denoted by OPND1 of the character specified by OPND3 is replaced by the character specified by OPND4.
COPX5 is present only if the 'tallying' option is set in T. The tally is stored in the field denoted by COPX5 when the examination is complete.

The options of the operation and the presence of each of the operands other than OPND1 are specified in the one-byte parameter T, which is coded as follows:

T0 T1 specifies whether all or part of the field denoted by OPND1 is to be examined. If T0 is 0, the setting of T1 is ignored and the whole of the field denoted by OPND1 is examined. If T0 is 0, OPND2 is not present.

If T0 is 1, the field denoted by OPND1 is split into three subfields by the first occurrence of the character specified by OPND2; the first part, the separating character, and the last part. If the separating character is the first character in the field, the first part is empty. If the separating character does not occur, or occurs only as the last character in the field, the last part is empty. T1 specifies the part of the field to be examined, the first part if T1 is 0, the last part if T1 is 1. If the part of the field to be examined is empty, no replacing is done and if tallying is specified, the field denoted by COPX5 is set to zero.

T2 specifies whether all character positions examined are tallied or replaced, or only those containing a particular value. If T2 is 0, all characters are tallied or replaced and OPND3 is not present. If T2 is 1, only character positions containing the character specified by OPND3 are tallied or replaced.
T3 T4 specifies whether tallying, replacing or both are required. If T3 T4 is 00, it is an error, and execution of the program is terminated. If T3 is 1, tallying takes place and COPX5 is present. At the end of the operation the tally is stored in the field denoted by COPX5. If T4 is 1, replacing takes place and OPND4 is present.

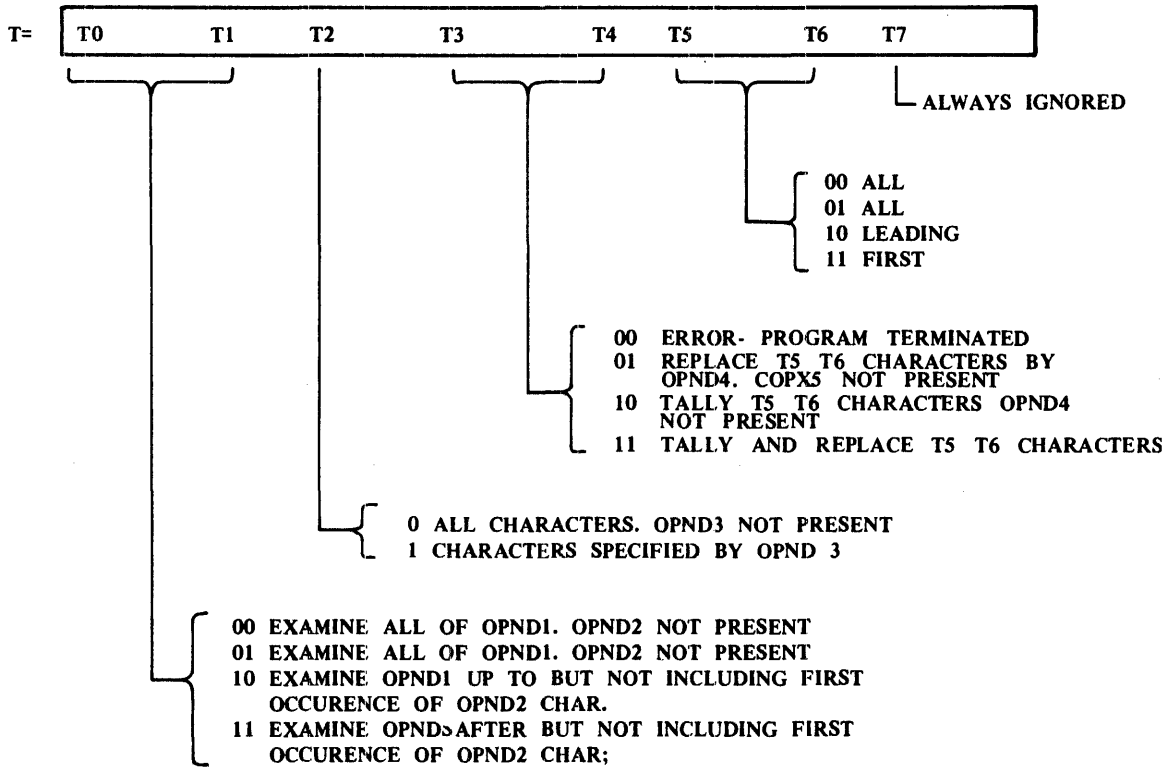
T5 T6 specifies the extent of the tallying or replacing within the part of the field examined. If T5 is 0, every character is eligible for tallying or replacing. If T5 is 1 and T2 is 0, it is an error and execution of the program is terminated. If T2 T5 T6 is 110, only occurrences of the character specified by OPND3 which occur before any other character in the part of the string examined are tallied or replaced. If T2 T5 T6 is 111, only the first occurrence in the part of the string examined of the character specified by OPND3 is tallied or replaced.

In all cases, the field is examined from left to right.

Any of the fields can overlap in any way. The characters specified by OPND2, OPND3 and OPND4 are accessed before any replacement takes place. The tally is stored in the field denoted by COPX5 after all other operations are complete. Refer to the diagram which follows.

Move Sign

Format: MVSGN OPND1 COPX2
Function: Copy the sign represented in the field OPND1 to the field denoted by COPX2. If OPND1 is unsigned, a positive sign is transferred to COPX2. If COPX2 is unsigned, no data movement takes place. Any non-standard sign representations are transferred if both fields have the same format, sign-type and sign position.



BRANCHING OPERATIONS

Branch Unconditionally

Format: BIS BADIS1 or BIL BADIL1 or
BES BADES1 or BEL BADEL1
Function: Obtain the next instruction from the location specified by BADDR1.

Branch on Overflow

Format: BOIS BADIS1 or BOIL BADIL1 or
BOES BADES1 or BOEL BADEL1
Function: If the overflow flag is set, obtain the next instruction from the location specified by BADDR1. Otherwise continue with the next sequential instruction. The overflow flag is not changed.

Branch No Overflow

Format: BNIS BADIS1 or BNIL BADIL1 or
BNES BADES1 or BNEL BADEL1
Function: If the overflow flag is not set, obtain the next instruction from the location specified by BADDR1. Otherwise continue with the next sequential instruction. The overflow is not changed.

Perform Enter

Format: PERIS K, BADIS1, or PERIL K, BADIL1 or PERSE K, BADES1 or PEREL K, BADEL1
Function: Construct an entry at the head of the Perform stack in the following format:

	K	Segment Number	Displacement	Line count
Bits	8	8	16	16

Obtain the one-byte value K from this instruction. SEGNO is the number of the current code segment. DISP is the displacement of the next sequential instruction relative to the current segment base. Adjust the stack pointer to point to the next possible entry. Obtain the next instruction from the location specified by BADDR1.

Alter

Format: ALIS BADIS1, DADDR2 or ALIL BADIL1, DADDR2 or ALES BADES1, DADDR2 or ALEL BADEL1, DADDR2

Function: Form a branch address of type BADEL from BADDR1 and copy it to the location specified by DADDR2. DADDR is a sixteen bit binary value which specifies a byte displacement relative to the base of data segment zero.

Enter

Format: NTRIS BADIS1 or NTRIL BADIL1 or NTRES BADES1 or NTREL BADEL1
Function: This is the same as PERFORM ENTER with K assumed to be zero.

Perform Exit

Format: PXIT K
Function: Compare the one byte value K with the K in the top entry of the perform stack. If they are unequal or the stack is empty, continue with the next S-instruction. If they are equal adjust the stack pointer to point to the previous entry, obtain the address of the next S-instruction from the information contained in the removed stack entry and replace the value in the line count register.

Go To Depending

Format: GTDI COPX1, N, BADILO,....BADILN or GTDE COPX1, N, BADELO,....BADELN
Function: Convert the value of the variable denoted by COPX1 to binary and compare it with the binary value N in the one byte field N. If the value of COPX1 is less than or greater than N, transfer control to the address specified by BADDRO. Otherwise use the value as an index to select one of BADDR1,....BADDRN and transfer control to the address so specified. If N is zero, control is transferred to the address specified by BADDRO.

Set Overflow

Format: OFV
Function: Set the overflow flag. The overflow flag is also set when divide by zero is encountered

and when the result of an arithmetic operation contains more significant digits than can be stored in the field provided for it.

Clear Overflow

Format: COFV
Function: Clear the overflow flag.

Altered Go To Paragraph

Format: GOPAR DADDR1
Function: Obtain the address of the next instruction from the location specified by DADDR1.

DADDR1 is a two byte binary value which specifies a byte address relative to the base of data segment zero. The instruction address held at this location is in the same format as a BADEL.

Exit

Format: EXIT
Function: The function is the same as PERFORM EXIT with K assumed to be zero.

Branch if Indicator Off

Format: BOFIS, IND1, BADIS2
BOFIL, IND1, BADIL2
BOFES, IND1, BADES2
BOFEL, IND1, BADEL2
Function: IND1 contains the offset of a digit within data segment zero. If this digit contains the value zero, obtain the next instruction from the location specified by BADDR2; otherwise continue with the next sequential instruction.

Branch if Indicator On

Format: BONIS, IND1, BADIS2
BONIL, IND1, BADIL2
BONES, IND1, BADES2
BONEL, IND1, BADEL2
Function: IND1 contains the offset of a digit within data segment zero. If this digit contains any value other than zero, obtain the next instruction from the location specified by BADDR2; otherwise continue with the next sequential instruction.

PERFORM ENTER LONG

Format: PERLIS N, BADIS1
PERLIL N, BADIL1
PERLES N, BADES1
PERLEL N, BADEL1

Function: Identical to PERFORM ENTER, except that K is generated as a two byte value from the two byte parameter N.

PERFORM EXIT LONG

Format: PXITL N

Function: Identical to PERFORM EXIT, except that K is compared as a two byte value with the two byte parameter N.

COMPARISON OPERATIONS AND OPERANDS

With the exception of the operation 'Compare for Class', CMPC, each comparison operation can be either a Conditional Branch or a Conditional Indicator Setting Operation. CMPC is always a Conditional Branch Operation. All comparison operations have the general format OPCODE, COMPARANDS, RV, PARAM. The format of PARAM is either a BADDR, in the case of a Branch Variant, or a list of zero, one, two or three indicators in the case of an Indicator Setting Variant. The type of operation and the Indicator Setting Requirements or the Condition for Branching and the Branch Address Variant are specified in the one byte field, RV. In operations other than CMPC, bit 3 of RV is used to discriminate between the branch and Indicator Setting Variants.

Conditional Branches (Except CMPC)

Bit 3 of RV is zero and PARAM is a BADDR. The two operands are compared and the result (first operand greater than, equal to or less than the second operand) is in turn compared with the relation being tested. If the Relational Condition is met, control is transferred to the address (BADDR) given in the instruction. If the Relational Condition is not met, control is passed to the next S-instruction.

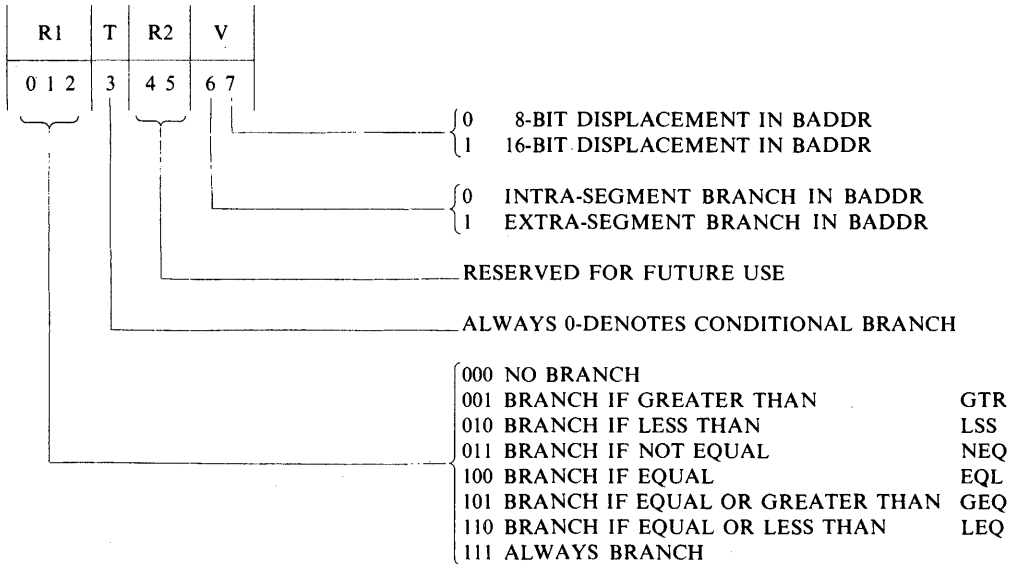
Bits 0, 1 and 2 of RV, which are referred to as R1, define the Relation.

Bit 3 of RV is zero, defining the operation as a Conditional Branch.

Bits 4 and 5 of RV must be zero. They are reserved for future use.

Bits 6 and 7 of RV, which are referred to as V, define the Address Variant.

The meanings of the possible values in RV are as follows:



The No-Branch and Always-Branch cases are included for completeness. They will not normally be generated by compilers.

Compare for Class Conditional Branch

The compare for class operation is always a Conditional Branch, irrespective of the value of B3. All three bits of R1 are ignored. Bits 3, 4 and 5 describe the Class Test to be performed. Bits 6 and 7 describe the type of Branch Address.

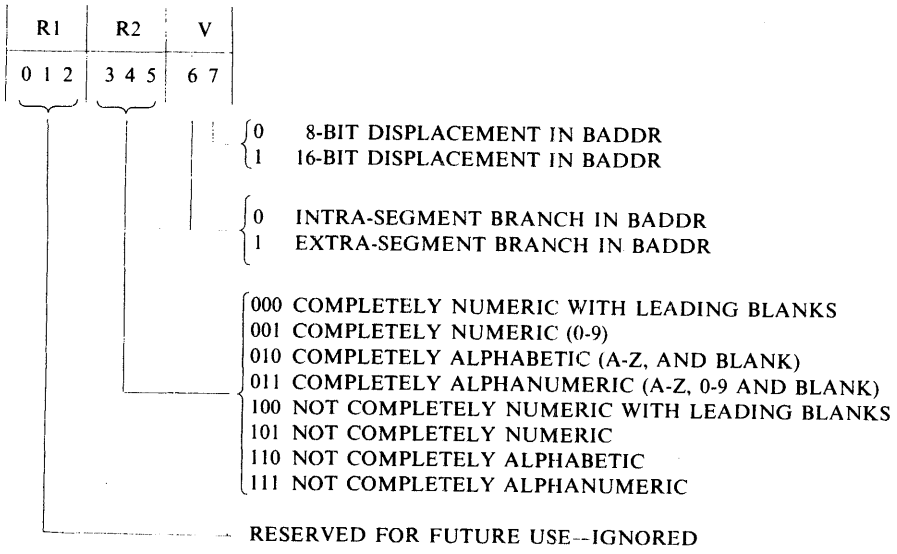
Conditional Indicator Setting

In a conditional indicator setting operation, bit 3 of RV is one and PARAM is zero, one, two or three bytes, containing nothing, or IND1, or IND1,IND2 or IND1,IND2,IND3.

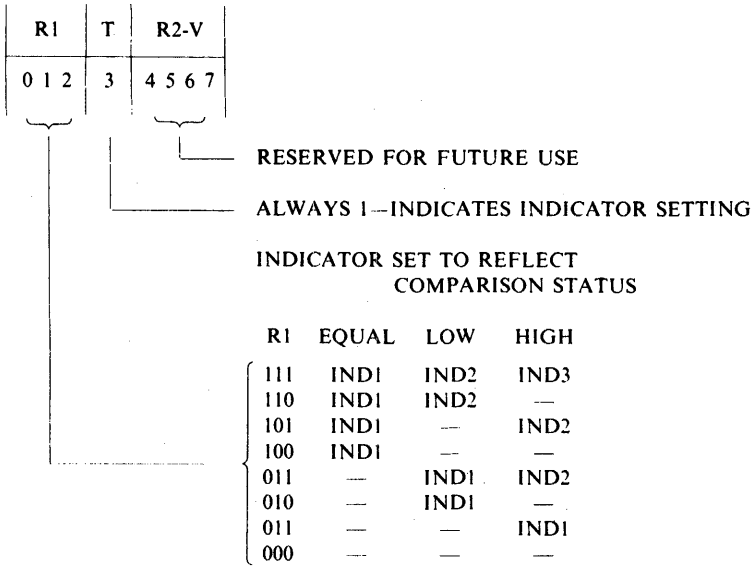
Bits 0, 1 and 2 of RV, known as R1, indicate how many of IND1,IND2 and IND3 are present and the action to be taken after the comparison has been performed. PARAM contains one byte (containing an IND) for each bit in R1 which has the value one.

The two operands are compared and the result of the comparison determined (first operand greater than second, high; first operand less than second, low; equality of operands, equal). The digits whose digit offsets in data segment zero are defined in the IND fields which are present, are each set to zero.

One of these digits can then be set to one to indicate the result of the comparison. The digit which is set is indicated in the figure which follows. Any two or all three of IND1, IND2 and IND3 can define the same digit. If the condition for setting a digit is realized, it is always set, despite any indication that it should remain zero because of the non-realization of some other condition. Where no IND is shown for a condition in the following figures and that condition is realized, no digit is set to one. The meanings of the possible values in RV are as follows:



The meanings of the possible values in RV for a conditional indicator setting operation are as follows:



R1	EQUAL	LOW	HIGH
111	IND1	IND2	IND3
110	IND1	IND2	—
101	IND1	—	IND2
100	IND1	—	—
011	—	IND1	IND2
010	—	IND1	—
011	—	—	IND1
000	—	—	—

A compiler will not normally generate the case R1 = 000 but it is defined for completeness.

Compare Alphanumeric

Format: CPA OPND1,OPND2,RV3,PARAM4
Function: Compare the two operand fields according to their binary value character by character from left to right. The first pair of unequal characters to be encountered determines the result of the comparison. The operand which contains the character which is positioned later in the collating sequence is the greater. Both operands must be in 8-bit unsigned format. If either is found to be in any other format, execution of the program is terminated. If the operands are the same length and all the character pairs are equal, the operands are equal. If the operands are of different lengths, the result of the comparison is the same as if the shorter operand were extended on the right to the length of the longer with space characters.

Compare for Spaces

Format: CPS OPND1,RV2,PARAM3
Function: Compare the contents of the field denoted by OPND1 and a second field of the same length containing all spaces according to the rules for compare alphanumeric.

Compare Numeric

Format: CPN OPND1, OPND2, RV3, PARAM4
Function: Compare the contents of the two operand fields according to their algebraic value. When the field sizes differ, consider the shorter extended to the length of the longer with leading zeros.

There are no restrictions on data type. When comparing 8-bit characters, only the digit part is considered except if the field has an overpunched sign in that character, in which case resolution of the sign/digit is carried out first to obtain the corrected digit value.

The interpretation of signs is as for arithmetic operations. Unsigned fields are considered positive. A field with magnitude zero and a negative sign participates in the comparison as if it had a positive sign.

The validity of the digit codes is not checked.

A field which is specified to be longer than 15 digits is reduced to 15 digits by truncation at the most significant end.

Compare for Zeros

Format: CPZ OPND1, RV2, PARAM3
Function: Compare the contents of the field denoted by COPX1 and a second field containing all zeros according to the rules for compare numeric. There is no restriction on the length of the field.

Compare Repeat

Format: CRPT, OPND1, OPND2, RV3, PARAM4
Function: Compare, according to the rules for compare alphanumeric, a field constructed by concatenating the field denoted by OPND1 with itself a sufficient number of times to make its length equal to the length of the field denoted by OPND2, with the field denoted by OPND2.

If the length of the field denoted by OPND2 is not an integral multiple of the length of the field denoted by OPND1, the rightmost copy of the field denoted by OPND1 is truncated on the right to give two equal length fields for comparison.

Compare for Class

Format: CMPC OPND1 RV2 BADDR3
Function: Determine whether the field denoted by OPND1 contains characters of the class specified by the R2 subfield of RV2. If the class condition is satisfied, transfer control to the address specified by BADDR3.

Alphabetic characters are the letters A to Z and space. Numeric characters are the digits 0 to 9.

The field denoted by OPND1 must be in 8-bit format. For the numeric or alphabetic and alphabetic tests it must be unsigned. For the numeric tests it may be signed, but will then only be classed as numeric if a valid sign code is present in the correct position.

If the data type of the field denoted by OPND1 is not one of those explicitly permitted above, the program will be terminated.

MISCELLANEOUS OPERATIONS

Communicate

Format: COMM OPND1
Function: Pass the information from the field denoted by OPND1 to the Master Control Program. Communicate message formats and functions are described in Section 5. The use of any message format other than those described causes termination of the program.

Fetch Communicate Response

Format: FCR COPX1
Function: Place the communicate response to the most recent communicate in the field denoted by COPX1 according to the rules for MVA. The response is three characters, the meaning of which is defined in Section 5. The field denoted by COPX1 is assumed to be unsigned 8-bit format.

Translate Communicate Response

Format: TCR COPX1
Function: Translate the communicate response to the most recent communicate according to the following rules and store the result in the field denoted by COPX1.

The field denoted by COPX1 must be 8-bit unsigned. The transformed response is stored, left justified in the field with truncation or zero fill on the right if necessary. The Master Control Program response consists of three bytes of information and is described in table 5-2.

The three bytes are referred to as byte 0, byte 1 and byte 2. The transformed response consists of 7 decimal digits, numbered from T1, the leftmost, to T7.

Byte 0 is ignored.

Bits 0-3 of byte 1 are moved to T1.

If bit 4 of byte 1 is zero, bits 0-3 of byte 2 are moved to T2, or else T2 is set to zero. Byte 2, considered as an 8-bit binary number, is converted to decimal and placed in the three decimal digits T3, T4 and T5.

If bit 5 of byte 1 is zero, T6 is set to zero, else T6 is set to 1.

If bit 6 of byte 1 is zero, T7 is set to zero, else T7 is set to 1.

Convert to Binary

Format: CDB OPND1 COPX2
Function: Convert the operand denoted by OPND1 from a decimal value to an unsigned 24 bit number. This is placed right justified (least significant end) in the field denoted by COPX2. If field COPX2 is larger than three bytes it is zero filled at the most significant end. If field COPX2 is smaller than three bytes, high order truncation is carried out. The number is assumed to be positive. Limited sign handling is applied when using signed fields for COPX2. For 4-bit signed and 8-bit with separate sign the sign is set positive. For an 8-bit overpunched sign, the sign is ignored and the field treated as 8-bit unsigned.

Convert to Decimal

Format: CBD OPND1 COPX2
Function: Convert the operand denoted by OPND1 from a binary value to a positive decimal number. This is placed right justified in the field denoted by COPX2 according to the rules for MVN.

The sign, if any, of the binary field is ignored. The binary value is extended with zeros or truncated on the left to 24 bits before conversion.

Set Line Count Register

Format: SLCR, N
Function: Set the line-count register to the two byte binary value N. The line-count register is kept by the interpreter and used for reporting the source line location of run-time errors. The register is 16 bits long. On entry to the program the register contains zero.

Increment Line Count Register

Format: ILCR
Function: Add 1 to the current contents of the line-count register. No overflow indication is given to the program if overflow occurs.

Read Line Count Register

Format: RLCR COPX1
Function: Convert the binary contents of the line-count register to decimal and store this value in the field denoted by COPX1 according to the rules for MVN.

Monitor Byte

Format: FLASH K
Function: Present the one byte value, K, to the hardware monitor, if fitted to the system, or else treat as a null operation.

Suspend

Format: SUSP N
Function: Cause the MCP to display message N from the message dictionary and then terminate program execution. The 2-byte parameter, N, is interpreted as an index into the system message dictionary.

BIT AND INDICATOR MANIPULATION

Set Indicator On

Format: SETON IND1
Function: Move the value one into the single digit whose digit offset within data segment zero is specified by the one byte binary number denoted by IND1.

Set Indicator Off

Format: SETOF IND1
Function: Move the value zero into the single digit whose digit offset within the data segment zero is specified by the one byte binary number denoted by IND1.

Set Bits On

Format: BITON OPND1, COPX2
Function: If the fields denoted by OPND1 and COPX2 are not both of 8-bit unsigned format and length one byte, the execution of the program is terminated.

Each bit in the field denoted by COPX2 which corresponds to a bit having the value one in the field denoted by OPND1 is set to the value one, irrespective of its initial value. All other bits retain their initial value.

This is equivalent to performing an inclusive-or operation on the contents of the two fields and storing the result in the second.

Set Bits Off

Format: BITOF, OPND1, COPX2
Function: If the fields denoted by OPND1 and COPX2 are not both of 8-bit unsigned format and length one byte, the execution of the program is terminated.

Each bit in the field denoted by COPX2 which corresponds to a bit having the value one in the field denoted by OPND1 is set to the value zero, irrespective of its initial value. All other bits retain their initial value.

Test Bits

Format: TESTB OPND1, OPND2, IND1, IND2, IND3
Function: If the fields denoted by OPND1 and OPND2 are not both of 8-bit unsigned format and length one byte, the execution of the program is terminated.

The three single digit fields whose digit offsets within data segment zero are denoted by IND1, IND2 and IND3 are each set to the value zero. The bits in the field denoted by OPND2 which correspond

to bits in the field denoted by OPND1 are then inspected. If one or more bits are inspected and all have the value one, the field denoted by IND3 is set to the value one. If one or more bits are inspected and all have the value zero, the field denoted by IND1 is set to one. If more than one bit is inspected and the values zero and one are both found, the field denoted by IND2 is set to the value one. If none of the bits in the field denoted by OPND1 have the value one, none of the fields denoted by IND1, IND2 and IND3 will be set to the value one.

Any two or all three of IND1, IND2 and IND3 can reference the same digit in data segment zero. If such a multiple reference occurs, on completion of the execution of the operation the multiply-referenced field will contain the value one if any of the individual references have caused it to be given the value one.

The five fields of the operation may overlap in any way. The effect of the operation is as if the two source values, from the fields denoted by OPND1 and OPND2, are extracted before any changes are made to the three result fields, denoted by IND1, IND2 and IND3.

THE MPLII VIRTUAL MACHINE

INTRODUCTION

The MPLII Virtual Machine consists of four main data spaces, registers to administer them, some general registers, and a large repertoire of S-instructions.

DATA SPACES

The four main data spaces are as follows:

1. PROGRAM DATA SPACE. Data is put into data segments, numbered from 0 through 63. The size of each data segment varies from 1 byte through 65535 bytes. Data is therefore referenced by a two-part address; a 6-bit data segment number and a 16-bit offset. Data segment number 0 is distinguished from all others in that:

   a. it is locked in memory
   b. it holds the working stack
   c. it holds data descriptors to all program data, whether in this segment or in other data segments
   d. it contains a mechanism which dynamically allocates and retrieves data space within the segment

2. PROGRAM CODE SPACE. Code is put into code segments, numbered from 0 through 63. The size of each code segment may vary from 1 byte through 65535 bytes. Code is therefore referenced by a two-part address: an 8-bit code segment number (of which only six bits are used), and a 16-bit offset. Each code segment contains a table of offsets pointing to the start of code for each procedure within the segment. Therefore, to address code for a procedure, the code segment number, and the number of the procedure within this segment are required. The code consists of single byte S-operators with or without parameters. Parameters can be coded into the 8-bit instruction value or may follow in-line in the code segment. Further details follow.

3. CONTROL STACK. The control stack, referred to as 'CONTROL' is resident in the TCB. The size may be from 1 byte up to 65535 bytes. It is used by the MPLII Virtual Machine for two purposes:

a. to store an activation record for each procedure entry including the return address at procedure exit
b. to store the address, temporarily, of the last declared data item in a data structure.

4. MESSAGE REFERENCE AREA. The message reference area is present only if the MPLII program is an MCS. This space is located in the Program Parameter Area in the PCB. Each message reference is a 4-byte pointer to messages stored in the data communications buffer space (trans-mural area) in memory.

REGISTERS

The virtual machine registers are stored in the SIWA within the TCB. The list which follows has been sub-divided according to their functions.

REGISTERS TO ACCESS CODE:

PSN (Program Segment Number, 8-bit). Contains the code segment number of the currently executing S-instruction.

SPN (Segment Procedure Number, 8-bit). Contains the procedure number within PSN of the currently executing S-instruction.

PCA (Program Current Address, 16-bit). Contains the byte offset, relative to the start of PSN, of the currently executing S-instruction.

REGISTERS TO ACCESS CONTROL STACK:

CSP (Control Stack Pointer, 16-bit). Contains the byte offset of the current top of the control stack, relative to the bottom of stack.

REGISTERS TO ACCESS DATA STACK:

There are two sets of registers pointing to the data stack (that is, to data segment zero), according to the MODE of the virtual machine. The two sets are to generate descriptors in declaration mode, and to access data in process mode.

To generate descriptors:

NDA (Next Descriptor Address, 16-bit). Contains the offset into data segment zero of the place where the next descriptor is to be placed.

SOL (Start of Last, 16-bit). Contains the offset into data segment zero of the start of the last data space allocated on the data stack.

EOL (End of Last, 16-bit). Contains the offset into data segment zero of the end of the last data space allocated on the data stack.

SEGN (Segment Number, 8-bit). Contains the number of the data segment in which data is being declared.

To access data:

LVL (Level, 4-bit). Contains the textual (or lexicographical) level of the procedure of the currently executing S-instruction.

REG1 (Region 1, 4-bit). Contains the level of the most referenced region of code.

REG2 (Region 2, 4-bit). Contains the level of the next most referenced region of code.

DISP (Display, sixteen 16-bit registers). Each DISP register contains the offset into data segment zero of the base of the descriptors for the dynamically most recent invocation of a procedure at the corresponding level. See 'DATA STACK STRUCTURE' following for an illustration.

STA (Stack pointer, 16-bit). Contains the offset into data segment zero of the next available byte at the top of the stack.

NLD (Number of Local Descriptors, 8-bit). Contains the number of local descriptors, used to administer the allocation of space from the literal pool which immediately follows the descriptors; that is, the offset into data segment zero of the base of the literal pool is

DISP(LVL) + NLD X DESC.SIZE

where DESC.SIZE has a value 4, the descriptor size in bytes.

REGISTERS FOR DATA COMMUNICATIONS:

NMR (Next Message Reference, 16-bit). Contains the number of the next available entry in the message reference area.

OTHER REGISTERS:

MODE (Mode of virtual machine, 4-bit). Contains a value indicating whether the virtual machine is in PROCESS mode, DECLARATION (data) mode or DECLARATION (remap) mode.

CARRY (Carry, 16-bit). Contains the resulting value of some arithmetic operations.

FV (Fetch Value, 24-bit). Holds the fetch result of the previous MCP communicate operation.

DESCRIPTOR FORMAT

Each data descriptor occupies four bytes on the data stack, structured as follows, in MPLII terminology:

```
DECLARE     1     DESCRIPTOR      CHARACTER(4)
              2   D.SEG.TYPE      BIT(8)
                3 D.SEG           BIT(6)
                3 D.TYPE          BIT(2)
              2   D.SUBTYPE       BIT(8)
                3 DUMMY           BIT(1)
                3 D.BIT.POSN      BIT(3)
                3 D.BIT.LENGTH    BIT(4)
              2   D.ORIGIN        CHARACTER(2)
```

If D.SEG.TYPE = @FF@, then the descriptor is of type SELF-RELATIVE, and data is in D.ORIGIN (that is, within the descriptor itself). Otherwise the following interpretation is valid:

D.SEG gives the number of the data segment. If D.TYPE = 2, the data type is BIT and the data described is the D.BIT.LENGTH bits starting at bit D.BIT.POSN in the D.ORIGIN byte of the segment D.SEG.

If D.TYPE = 1, the data is type CHARACTER and the data described is the D.SUBTYPE bytes starting at the D.ORIGIN byte of segment D.SEG. If D.TYPE = 0 and D.SUBTYPE = 2, then the data is type FIXED and the data described is the two-byte value at D.ORIGIN in segment D.SEG. If D.TYPE = 0, and D.SUBTYPE = 1, then the type is MESSAGE.REFERENCE and the data described is the interpretation of D.ORIGIN by the MCP to give a message space in the data communications buffer pool (implementation dependent). All other values are combinations of D.TYPE and D.SUBTYPE are illegal.

DATA STACK STRUCTURE

At each textual (lexicographical level), the order of items on the data stack, from bottom upwards, is as follows:

1. descriptors to parameters
2. descriptors to local data
3. literal pool
4. local data space
5. working stack

Some of these items may not be present, but the relative order is preserved. The top of the working stack is indicated by STA, the base of each level is indicated by DISP(LVL). This is illustrated in figure 7-6, where the base of the stack is drawn at the bottom of the figure.

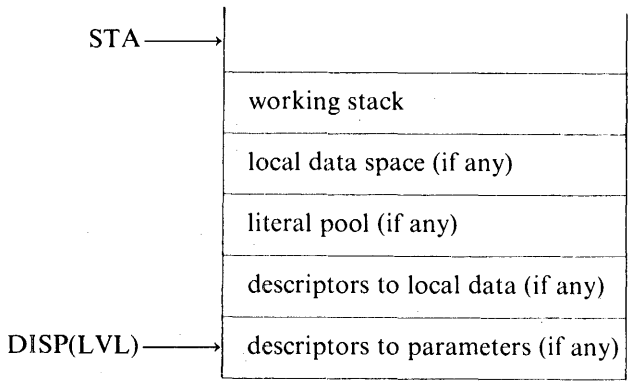


Figure 7-6. Data Stack Structure

An actual example is given in figure 7-7, corresponding to the MPLII skeleton program given in figure 7-8. The stack is shown at the point in the program after the declaration of D in procedure R (line 6). The stack is drawn with the base at the bottom of the figure.

In the example program, the outer level (procedure P) is at textual level 0, so that DISP(0) points to the base of the descriptors for data at this level. These are two self-relative descriptors for the FIXED items A and B.

Procedures Q and S are at textual level 1. The call on S at line 14 results in DISP(1) pointing to the base of descriptors at this level; that is, two self-relative descriptors for items E and F. However, the subsequent call at line 12 on Q, which is also at level 1, results in DISP(1) pointing at the base for

procedure Q. In fact, DISP(1) points to the parameter N. The actual parameter is B; therefore the descriptor corresponding to the formal parameter N is of type FIXED, pointing to the ORIGIN field in the Self-relative descriptor B. Following this is a descriptor to local data within Q; that is, C. As this is type CHARACTER, the descriptor points to data further up the stack (there is no literal pool in this example).

Procedure Q at line 8 calls procedure R which is at level 2, resulting in the setting up of register DISP(2). The formal parameter M in this case corresponds to the actual parameter C; the parameter descriptor is therefore of type CHARACTER pointing to the data for C.

```
PROCEDURE P;                          1
  DECLARE (A, B) FIXED;               2
  PROCEDURE Q(N);                     3
    DECLARE C CHARACTER(10);          4
    PROCEDURE R(M);                   5
      DECLARE D FIXED;                6
      :
    END R;                            7
    :
    R(C);                             8
    :
  END Q;                              9
  :
  PROCEDURE S;                        10
    DECLARE (E, F) FIXED;             11
    :
    Q(B);                             12
    :
  END S;                              13
  :
  S;                                  14
  :
END P;                                15
FINI;                                 16
```

Figure 7-7. Example MPLII Program

THE INSTRUCTION SET

Each S-instruction occupies one byte, and may or may not take parameters. The parameters can be encoded into the S-instruction, or follow in-line. These parameters can each be one or two bytes in length. In defining a particular instruction, a mnemonic name is used to represent the operation and a bracketed list of numbered parameters defines the parameters. Parameters named by using an 'A' are one byte long; parameters named by using a 'B' are two bytes long.

For example, an operation requiring an in-line one-byte parameter might be called OP(AO) and one requiring one one-byte and two two-byte parameters might be called OP(AO,B1,B2).

Parameters encoded into the S-instruction are represented unbracketed. The instruction mnemonic indicates an encoded parameter; mnemonics ending in 'N' have a literal value encoded within them; those

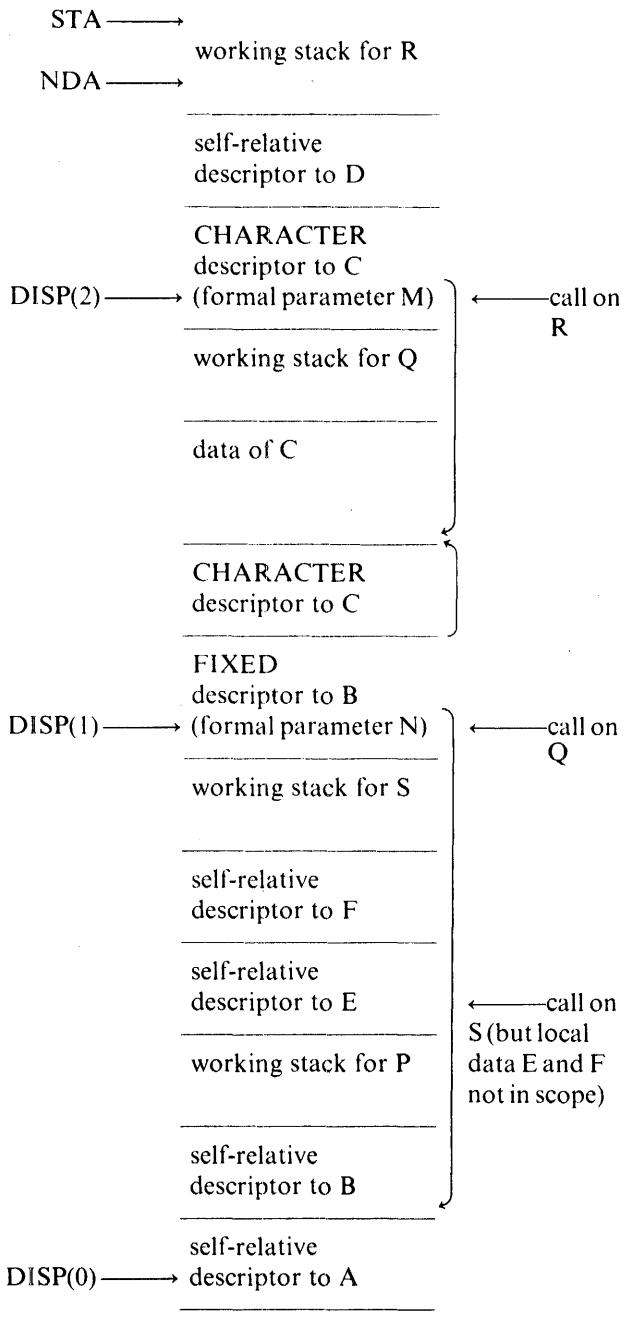


Figure 7-8. Example Data Structure

ending in 'R' have a coding of a reference to REG1 or REG2, and those ending in 'OR' have a coding of both an occurrence number (that is, the number of a descriptor relative to the base of a level) and a reference to REG1 or REG2.

The interpretation of the S-instruction 8-bit value depends on the MODE of the virtual machine. Table 7-4 gives the complete instruction set in PROCESS mode and table 7-5 gives the instruction set in DECLARATION (data and remap) mode.

Table 7-4. MPLII S-Instructions (Process Mode)

SECOND DIGIT	FIRST DIGIT 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		SNMI (A0,A1)	LSZR 0,(A0)	BNDXR 0,(A0)	LFVR 0,(A0)	LADR 0,(A0)	LADØR 24,0	LADØR 28,0	LADØR 0,0	LADØR 4,0	LADØR 8,0	LADØR 12,0	LADØR 10,0	LADØR 20,0	LITN 0	LITN 16
1		SNM2 (A0,A1)	LSZ (A0,A1)	BNDX (A0,A1)	LFV (A0,A1)	LAD (A0,A1)	LLV16 (B0)	LLV8 (A0)	LCY	FVAL	INDXØR 8,0	INDXØR 12,0	INDXØR 0,0	INDXØR 4,0	LITN 1	LITN 17
2		LTYP (A0,A1)	LSZR 1,(A0)	BNDXR 1,(A0)	LFVR 1,(A0)	LADR 1,(A0)	LADØR 24,1	LADØR 28,1	LADØR 0,1	LADØR 4,1	LADØR 8,1	LADØR 12,1	LADØR 16,1	LADØR 20,1	LITN 2	LITN 18
3					LLS (A0...AN)		CØMME (A0...AN)	CØMM (A0...AN)			INDXØR 8,1	INDXØR 12,1	INDXØR 0,1	INDXØR 4,1	LITN 3	LITN 19
4			LDA (A0,A1)	LSAR 0,(A0)	LØAR 0,(A0)	INDX 0,(A0)	LADØR 25,0	LADØR 29,0	LADØR 1,0	LADØR 5,0	LADØR 9,0	LADR 13,0	LADØR 17,0	LADØR 21,0	LITN 4	LITN 20
5			LTA (A0,A1)	LSA (A0,A1)	LØA (A0,A1)	INDX (A0,A1)	DECL (A0)	SRMP (A0,A1)	RMP (A0,A1,A2)	CØNV (A0)	INDXØR 9,0	INDXØR 13,0	INDXØR 1,0	INDXØR 5,0	LITN 5	LITN 21
6	LIB (A0,A1,A2)	LSN (A0,A1)	DVAL	LSAR 1,(A0)	LØAR 1,(A0)	INDXR 1,(A0)	LADØR 25,1	LADØR 29,1	LADØR 1,1	LADØR 5,1	LADØR 9,1	LADØR 13,1	LADØR 17,1	LADØR 21,1	LITN 6	LITN 22
7	GØBL (B0)	GØBS (A0)	GØFL (B0)	GØFS (A0)	GØIL (B0)	GØIS (A0)	GØUL (B0)	GØUS (A0)	DUPS	RTDS	INDXØR 9,1	INDXØR 13,1	INDXØR 1,1	INDXØR 5,1	LITN 7	LITN 23
8	CALL (A0,A1)	EXIT	ADDD	SUBD	ADD	SUB	LADØR 26,0	LADØR 30,0	LADØR 2,0	LADØR 6,0	LADØR 10,0	LADØR 14,0	LADØR 18,0	LADØR 22,0	LITN 8	LITN 24
9	CALS (A0)	RTRN	ADDN 1	SUBN 1	ADDDN 1	SUBDN 1	NEG	MULT	DIV	MØD	INDXØR 10,0	INDXØR 14,0	INDXØR 2,0	INDXØR 6,0	LITN 9	LITN 25
A	NTR (A0...A4)	EXITF (A0,A1)	CASS (A0...A1)	DADD2	ADDDN 2	SUBDN 2	LADØR 26,1	LADØR 30,1	LADØR 2,1	LADØR 6,1	LADØR 10,1	LADØR 14,1	LADØR 18,1	LADØR 22,1	LITN 10	LITN 26
B	NTRS (A0...A2)	RTNF (A0,A1)	CASL (A0,B1...BN)	LIL	STD	STN	STLS (A0...AN)	FILL (B0...AN)	SBS2	SBS3	INDXØR 10,1	INDXØR 14,1	INDXØR 2,1	INDXØR 6,1	LITN 11	LITN 27
C	AND		SRL	NEQN 0	EQUN 0	STON 0	LADØR 27,0	LADR 31,0	LADØR 3,0	LADØR 7,0	LADØR 11,0	LADØR 15,0	LADØR 19,0	LADØR 23,0	LITN 12	LITN 28
D	IØR	GTRN 1	SLL	NEQN 1	EQUN 1	STDN 1		FF8	FF16	FF24	INDXØR 11,0	INDXØR 15,0	INDXØR 3,0	INDXØR 7,0	LITN 13	LITN 29
E	EØR		NØT	NEQN 2	EQUN 2	STON 2	LADØR 27,1	LADØR 31,1	LADØR 3,1	LADØR 7,1	LADØR 11,1	LADOR 15,1	LADØR 19,1	LADØR 23,1	LITN 14	LITN 30
F	TFN	GTR	LSS	NEQ	EQU	GTE	LSE	SSL (A0)	LSL (A0)	SBSG (A0,A1)	INDXØR 11,1	INDXØR 15,1	INDXØR 3,1	INDXØR 7,1	LITN 15	LITN 31
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

PROCESS MODE INSTRUCTIONS

The instructions in PROCESS mode can be divided into three major and two minor groups. The major groups are:

1. descriptor loading instructions
2. value storing instructions
3. instructions to control the flow of control

The minor groups are:

4. instructions which communicate with the MCP
5. interpreter library call
6. instructions to start declaration mode execution

Table 7-5. MPLII S-Instructions (Declaration Mode)

SECOND DIGIT	FIRST DIGIT 0	1	2	3	4	5	6	7	8
0		CH (A0)	CHN 0	CHN 16	BITS (A0...AN)				
1	SRN 1	CHNM 1,(A0)	CHN 1	CHN 17	MR				
2	SRN 2	CHNM 2,(A0)	CHN 2	CHN 18	MRAF (A0)				
3	SRN 3	CHNM 3,(A0)	CHN 3	CHN 19	MRAD (A0,A1)				
4	SRN 4	CHNM 4,(A0)	CHN 4	CHN 20	DEND				
5	SRN 5	CHNM 5,(A0)	CHN 5	CHN 21					
6	SRN 6	CHNM 6,(A0)	CHN 6	CHN 22					
7	SRN 7	CHNM 7,(A0)	CHN 7	CHN 23					
8	SR (A0)	DCH (A0,A1)	CHN 8	CHN 24					
9	FIXDN 1	FIXD (A0)	CHN 9	CHN 25					
A	FIXDN 2	NST	CHN 10	CHN 26					
B	FIXDN 3	UNST	CHN 11	CHN 27					
C	FIXDN 4	USTD (A0)	CHN 12	CHN 28					
D	BND (B0)	USTR (A0)	CHN 13	CHN 29					
E	BNDS (A0)	DUM (A0)	CHN 14	CHN 30					
F	DBND (A0,A1)	DDUM (A0,A1)	CHN 15	CHN 31					
	0	1	2	3	4	5			

LOAD INSTRUCTIONS

The load instructions subdivide into:

1. those which load actual values (self-relative descriptors)
2. those which load addresses (FIXED, BIT and CHARACTER descriptors)

Load Value Instructions

1. LIT16(B0)
   LIT8(A0)
   LITN

These instructions load a self-relative descriptor containing literal values onto the working stack. The literal is either a 16-bit value, an 8-bit value, or a value encoded into the instruction.

2. EQU

EQUN
NEQ
NEQN
LSS
LSE
GTE
GTR
GTRN

These instructions leave a self-relative descriptor with a value 0 or 65535, depending on whether the comparison was true or false. EQUN, NEQN and GTRN all compare the fixed value of the top-of-stack descriptor with a value encoded in the instruction, and replace that descriptor with the comparison result. The others compare the data described by the top two descriptors and replace them by the result. The CHARACTER descriptors are compared left-to-right with space filling but other types have their fixed values compared.

3. LFV(A0,A1)
   LFVR(A0)
   FVAL

These instructions load a self-relative descriptor with the fixed value of some other descriptor. This descriptor is either at level A0, occurrence A1(LFV), or at level given by REG1 or REG2 depending on the coding of R, occurrence A0(LFVR), or at the top of stack(FVAL).

4. NOT
   NEG

These instructions replace the top descriptor by a self-relative descriptor with a value of the logically inverted (NOT) or arithmetically negated (NEG) fixed value of that descriptor.

5. LCY

This instruction loads a self-relative descriptor with the value of the CARRY register, and also clears CARRY.

6. FF8
   FF16

These instructions load a self-relative descriptor with the values of the most significant 8-bits (FF8) or the least significant 16-bits (FF16) of the FV register.

7. LSN(A0,A1)
   LSZ(A0,A1)
   LSZR(A0)
   LTYP(A0,A1)

These instructions load the segment number (LSN), a value corresponding to the size (LSZ), or a value corresponding to the type (LTYP) of the descriptor at level A0, occurrence A1. LSZR is the same as LSZ except that the descriptor is found at the level corresponding to REG1 or REG2, occurrence A0.

8. TFN

This instruction loads a self-relative descriptor with value 0 or 65535 depending on whether the top-of-stack descriptor is a null message-reference or not.

Load Address Instructions

1. LAD(A0,A1)
   LADR(A0)
   LADOR

These instructions load a descriptor (either FIXED, BIT or CHARACTER) to top-of-stack, pointing to data described by a given descriptor. The given descriptor is found at level A0, occurrence A1(LAD) or at level given by REG1 or REG2, occurrence A0(LADR) or at level given by REG1 or REG2, and occurrence encoded in the instruction (LADOR). For all but self-relative descriptors, these instructions merely load a copy of that descriptor: for self-relative descriptors, a type FIXED descriptor is loaded pointing to the ORIGIN field of the self-relative descriptor.

2. INDX(A0,A1)
   INDXR(A0)
   INDXOR

These instructions use the fixed value of the top-of-stack descriptor to increment the origin field of the descriptor specified in the instruction and replace the top-of-stack with the indexed descriptor. The specified descriptor is given by level and occurrence in a similar manner to LAD (see 1. above).

3. BNDX(A0,A1)
   BNDXR(A0)

These instructions use the fixed value of the top-of-stack descriptor to increment the bit origin of the descriptor specified in the instruction, and replace the top-of-stack with the indexed descriptor. The specified descriptor is given by level and occurrence in a similar manner to LAD (see 1. above). Non BIT descriptors effectively have their ORIGIN field incremented by the fixed value divided by 8.

4. SBS2
   SBS3

These instructions implement the MPLII 'SUBSTRING' function with either 2 or 3 MPLII parameters. They use the top 2 or the top 3 descriptors on the stack, and replace them with a descriptor pointing to the required character field.

5. LLS(A0,A1...AN)

This instruction loads a descriptor of type CHARACTER, size A0 on the top-of-stack, and the literal string A1...AN (which is A0 bytes long) into the literal pool.

6. SNM1(A0,A1)
   SNM2(A0,A1)

These instructions implement the MPLII 'SETNAME' routine with either 2 or 3 parameters respectively. SNM1 sets the descriptor at level A0, occurrence A1 to the top-of-stack descriptor and removes the top-of-stack descriptor. SNM2 sets the descriptor at level A0, occurrence A1 to the second descriptor on the stack, and additionally sets the size field to the value of the top-of-stack descriptor, and removes the top two descriptors.

7. LIL

This instruction loads a descriptor on the top-of-stack with an illegal type field.

Descriptor Duplication and Removal

1. DUPS

This instruction places a copy of the top descriptor on the working stack.

2. RTDS

This instruction removes the top descriptor from the working stack.

Store Instructions

These instructions are divided into three groups:

1. those which assign values from one field to another
2. those which store literal values into data fields
3. those which perform some operation on data fields while updating one of those fields

Assignment

STN
STD

These instructions assign the value described by the top-of-stack descriptor into the data area addressed by the second descriptor on the stack. The instructions are illegal if the second descriptor is a value (self-relative). If both descriptors are type CHARACTER the instructions are illegal if the destination string starts within the source string. If both descriptors are type CHARACTER, character movement takes place with blank fill or truncation on the right: otherwise the assignment takes place as fixed values. STD removes both descriptors while STN removes only the top one, leaving the address on the top-of-stack.

Literal Storage

1. STDN

This instruction stores the value encoded in the instruction into the data area described by the top descriptor on the stack, and removes that descriptor.

2. STLS(A0,A1...AN)

This instruction stores into the data area described by the top descriptor a value determined by the type of that descriptor. If it is CHARACTER the A0 bytes in the string A1...AN are stored, otherwise the fixed value of A1 is stored.

3.FILL(B0,A1...AN)

This instruction moves the B0 bytes of the string A1...AN to the data field starting at the ORIGIN of the top descriptor.

4. FF24

This instruction moves the 3 bytes of the FV register to the data field described by the top descriptor, with space fill or truncation. If this descriptor is not of type CHARACTER the instruction is illegal.

Updating

1. ADD
   SUB
   MULT
   DIV
   MOD
   AND
   IOR

EOR
SRL
SLL

These instructions perform: the addition, subtraction, muliplication, division giving quotient, division giving remainder, logical-and, logical-or, logical-non-equivalence, shift right, or shift left operations on the fixed values of the two top descriptors, updating the second with the result and removing the top descriptor.

2. ADDD
   SUBD

These instructions perform the same operation as ADD and SUB except that the top two descriptors are removed.

3. ADDN
   SUBN

These instructions perform the addition or subtraction operation on the fixed value of the top descriptor using a value encoded in the instruction, and leaving the top descriptor on the stack.

4. ADDDN
   SUBDN

These instructions perform the same operation as ADDD and SUBD except that the top descriptor is removed.

5. CONV(A0)

This instruction implements the MPLII 'CONVERT' routine using coding derived from A0. The convert source is the fixed value of the top-of-stack descriptor, and the convert destination is the character string described by the second descriptor on the stack (if this descriptor is not of type CHARACTER the instruction is illegal).

6. DADD2

This instruction implements the MPLII 'DEC.ADD' routine with 2 parameters; the decimal value given by the top descriptor is added to the decimal value given by the second descriptor and both descriptors are removed.

Control Instructions

These instructions are divided into two groups:

1. those which transfer control to some other part of the current procedure
2. those which transfer control to some other procedure

Transfer within Procedures

1. GOBL(B0)
   GOBS(A0)
   GOFL(B0)
   GOFS(A0)

These instructions cause transfer of control to the S-instructions either forward (GOFL and GOFS) or backward (GOBL and GOBS) by the number of bytes given either by A0 (short) or B0 (long).

2. GOIL(B0)
   GOIS(A0)
   GOUL(B0)
   GOUS(A0)

These instructions cause a conditional go forward (either A0 or B0 bytes) either if (GOIL and GOIS) or unless (GOUL and GOUS) the fixed value of the top-of-stack descriptor is odd; that is, 'true'.

3. CASS(A0,A1...AN)
   CASL(A0,B1...BN)

These instructions cause a go forward by the number of bytes indicated by one of A1...AN(CASS) or by one of B1...BN(CASL) indexed by the fixed value of the top-of-stack descriptor. If this value is zero, control continues at the next instruction; if it is between 1 and A0, transfer is made to the instruction addressed by the parameter numbered with that value, and if the value is outside these values, transfer is to the instruction addressed by the last value.

Transfer Outside a Procedure

1. CALL(A0,A1)
   CALS(A0)

These instructions execute a procedure call. They save a return address on CONTROL and set PSN and PCA to address the entry instruction of the required procedure. CALL uses A0 as the new PSN and CALS assumes the current PSN. The remaining parameter is the new SPN and indexes segment PSN's procedure table to obtain a value for PCA.

2. NTR(A0,A1,A2,A3,A4)
   NTRS(A0,A1,A2)

These instructions are the first instruction of every procedure. They build an activation record and save it on CONTROL and set up registers for correct stack and declaration administration.

For NTR, A0 is the level of the new procedure, A1 encodes the values for REG1 and REG2 registers, A2 is the number of formal parameters, A3 is the number of local declarations and A4 is the size of the literal pool. NTR causes a switch of the the virtual machine to declaration mode.

For NTRS, A0 is the level of the new procedure, A1 encodes the values for REG1 and REG2 registers, and A2 is the size of the literal pool (the number of formal parameters and local declarations is assumed to be zero). NTRS continues the virtual machine in process mode.

3. EXIT
   XITF(A0,A1)
   RTRN
   RTNF(A0,A1)

These instructions cause exit from a procedure. Exiting involves removing the latest activation record from CONTROL in order to reset volatile registers. A return address is unstacked from CONTROL to reset PSN, SPN and PCA. EXIT performs the exiting operation. RTRN unstacks the top descriptor, performs an EXIT, then stacks the descriptor again. XITF performs repeated exits until it has exited from the procedure A1 in segment A0. RTNF preserves and restores a descriptor as in RTRN but performs an XITF instead of an EXIT.

Communication with the MCP

1. COMM(A0...AN)
   COMME(A0...AN)

These instructions transfer control to the MCP with a message constructed from values encoded in A0...AN and data described on the working stack. The CPA is constructed by the virtual machine from

these values. If an error is encountered by the MCP in any of the data or is reported back via a value of hexadecimal 80 in the most significant byte of FV, then the virtual machine will give the program a fatal error if the instruction was COMM, or allow the program to continue if the instruction was COMME.

2. SSL(A0)

This instruction sets the size of data segment number A0 to the fixed value of the top-of-stack descriptor so long as the segment is not present and the size is reduced.

3. LSL(A0)

This instruction loads a self-relative descriptor with value equal to the size of segment A0.

4. SBSG(A0,A1)

This instruction loads a descriptor of a subfield of the FPB data segment corresponding to the FIB whose number is the fixed value of the top descriptor. The field described starts at offset A1 from the start of the segment and has type and size derived from the encoding of A0.

Interpreter Library Call

LIB(A0,A1,A2)

This instruction is interpreted differently by different implementations of the virtual machine. If the Interpreter recognizes the parameter A2, then a micro-coded library routine is executed; if not, then a call is made on the (S-code) procedure with PSN = A0 and SPN = A1.

Declaration Mode Entry

DECL(A0)
SRMP(A0,A1)
RMP(A0,A1,A2)

These instructions, as well as the NTR instruction, cause the virtual machine to switch to declaration mode. The first parameter is the number of the next descriptor to be declared and is used to set NDA. SRMP and RMP specify a data area (segment A1, or that area described by the descriptor at level A1, occurrence A2) which is to be remapped. The registers SOL, EOL and SEGN are initialized to the appropriate data areas and declaration mode (data or remap) is entered.

Declaration Mode Instructions

The instructions in DECLARATION mode can be divided into four groups:

1. those which generate descriptors in data segment zero
2. those which administer data space allocation
3. those which administer structured declarations
4. one which ends declaration mode

Descriptor Generation Instructions

1. SR(A0)
   SRN
   FIXD(A0)
   FIXDN

These instructions generate self-relative and FIXED descriptors. The number to be declared is either a parameter or encoded within the instruction.

2. CH(A0)
   CHN
   DCH(A0,A1)
   CHNM(A0)

These instructions generate CHARACTER descriptors. Except for CHNM, only one descriptor is generated, of size given either by a parameter (CH) or encoded in the instruction (CHN) or from the fixed value of the descriptor at level A0, occurrence A1 (DCH). CHNM generates A0 descriptors of size encoded in the instruction.

3. BITS(A0,A1...AN)

This instruction generates type BIT descriptors. Each parameter encodes either the position and lengths of the bit string in byte SOL or the amount by which SOL and EOL should be advanced. A zero parameter terminates the BITS declaration.

4. MR
   MRAF(A0)
   MRAD(A0,A1)

These instructions generate type MESSAGE.REFERENCE descriptors. Each of these declares one descriptor with origin NMR and allocates space in the message reference array by incrementing NMR by the number of allocated entries (1, A0, or the fixed value of the descriptor at level A0 and occurrence A1).

Data Space Allocation Instructions

1. BND(B0)
   BNDS(A0)
   DBND(A0,A1)

These instructions allocate space for arrays. They all set the register EOL to SOL plus a byte value: this value is either B0, or A0, or the fixed value of the descriptor at level A0 and occurrence A1. If this sum is less than EOL, the instruction is illegal.

2. DUM(A0)
   DDUM(A0,A1)

These instructions allocate space for dummy data items; that is, where no descriptor is required. They set SOL to EOL and increment EOL by the value A0 or the fixed value of the descriptor at level A0 and occurrence A1.

Structure Declarations

NST
UNST
USTD(A0)
USTR(A0)

These instructions enable the substructuring of data items by administering the saving and restoring of EOL. NST saves EOL on CONTROL and sets EOL to SOL. UNST resets EOL from that entry: if it is less than the current value of EOL, the instruction is illegal. USTD sets the size of the description at the current level, occurrence A0 to EOL minus the ORIGIN value of that descriptor, and discards the value stacked on CONTROL. USTR performs the same action as USTD except that it restores EOL from the stacked value.

Declaration Mode Termination

DEND

This instruction causes the virtual machine to switch to PROCESS mode.

SECTION 8
B 90 DEPENDENT IMPLEMENTATION

MEMORY MANAGEMENT

Memory management is divided into two distinct parts; slice management and segment management. Segment management is performed by the Virtual Memory (VM) Subsystem. Segments are regarded as being of relatively short term residency in memory, and unless marked as locked by a program are held in the overlay arena. Task run structures, I/O handlers, and certain MCP routines are regarded as relatively long term residents in memory and are stored in slice memory as slices. Memory is physically divided into a number of pages, each of which can contain up to 65536 bytes. Data and code are organized so that all the segments belonging to a slice are contained in the same page. Slices may not straddle page boundaries. However, a task may be scattered over many pages, insofar as the slices belonging to the task are resident in different memory pages. Page 0 is special and is organized differently from other pages. For example, it contains all the code for the MCP (refer to CCB later). No MCP code can currently reside in a non-zero page. Figures 8-1 and 8-1A show the layout of memory. The 4K ROM (in page 0) is a basic feature of the B 90 and contains routines for machine initialization at power on time, and bootstrap routines to load basic firmware from cassette or disk devices. The basic intrinsics and global locations are set up at initialization time and contain the basic memory and task management routines, the interrupt analysis routines and the basic pointers to key points within the memory structure. Slice memory is that portion of RAM currently used for slices. The boundary between slice memory and the overlay arena in any page is not fixed and floats as tasks are executed. The overlay arena is used for storing the currently used overlayable segments of code and data. Since code and data segments are not of fixed length and the overlay arena is not compressed, it is necessary to maintain information about available and used areas (refer to figure 8-3). The transmural-area (TMA) (in page 0) is used to hold information about the physical peripherals; that is, the peripheral handling tables (PHTs). These tables are used in performing all I/O operations and are built at initialization time. They are permanently resident.

SLICES

Figure 8-2 shows more details of Page 0 Slice Memory. At the beginning (low-address end) of Slice memory is the slice address table and page address table. Each table consists of two-byte addresses. Each address in the first table is either zero for non-existent slice or points to the slice descriptor. Therefore, the table is known as the Slice Address Table (SAT). A slice number is the ordinal position in the SAT of the slice address. Twice the slice number gives a byte index into the SAT of the corresponding slice address. The second table, known as the Page Address Table (PAT) contains corresponding entries indicating the memory page which contains the slice and whether the slice is fixed in the page. Beyond the PAT is a table CCBDRIX of the Disk Directory Indices of the CCBs held from the entry FREESN in the SAT and beyond this is the first slice descriptor. Slice memory contains no unused space. Whenever a slice is removed, all succeeding slices are packed up and the area freed at the top of slice memory is added to the overlay arena. If space is required for slice memory, the required amount of space is obtained at the bottom of the overlay arena and is then added to slice memory. Slices are then slid up until the required space is in the correct position.

All slices currently known to the system must have a slice descriptor in slice memory. Each slice descriptor is pointed to by the corresponding entry in the SAT and PAT. The slice descriptors are chained according to their physical order in memory by means of a link in each descriptor to the next descriptor. The link in the last descriptor in the chain points to a byte containing hexadecimal 'FF'. This byte, known as the 'stopper' marks the boundary between slice memory and the overlay arena. The part of memory between the slice descriptor and the location pointed to by the descriptor link determines the current size of the slice in memory. (Note that this size will be zero for an absent slice.)

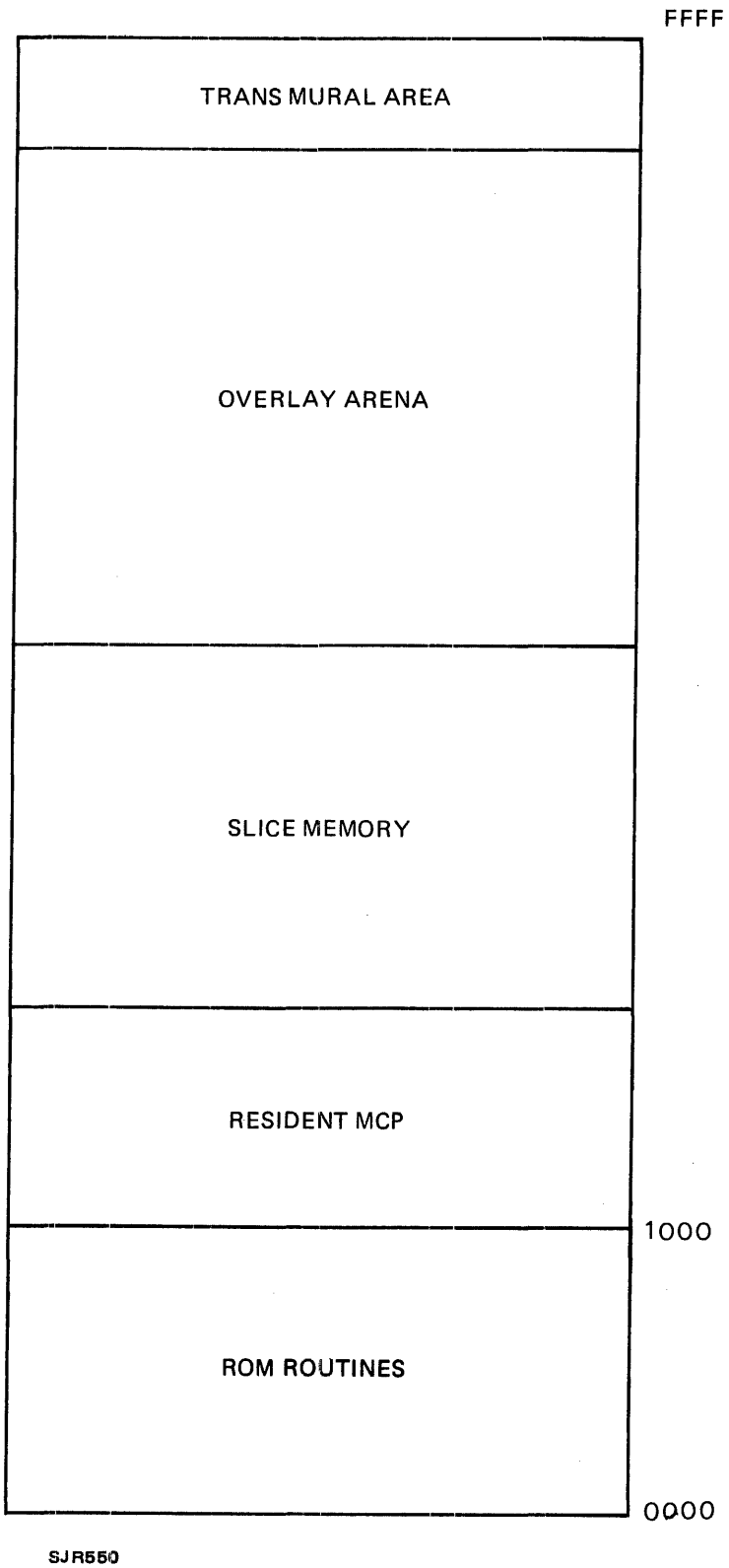


Figure 8-1. Page 0 Memory Layout

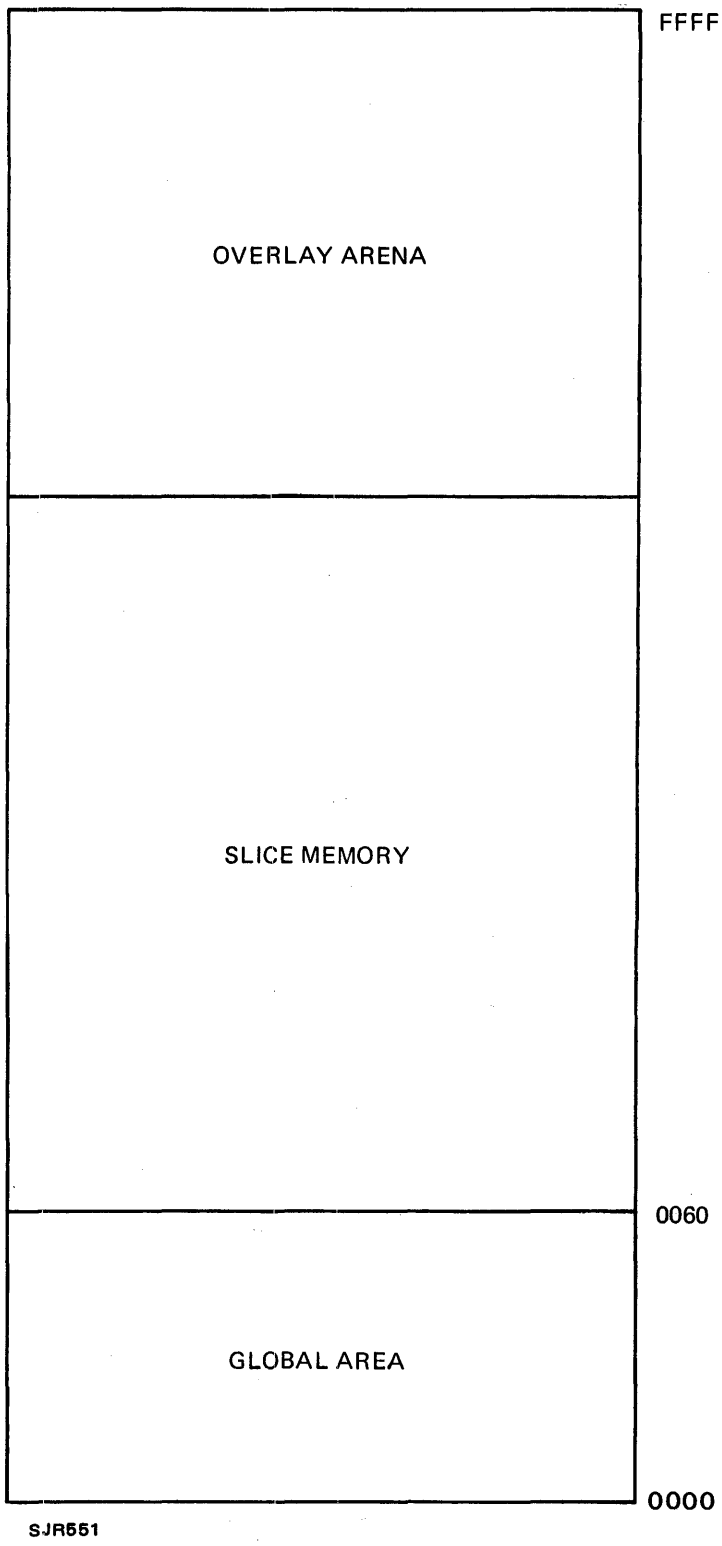


Figure 8-1A. Non-Zero Page Memory Layout

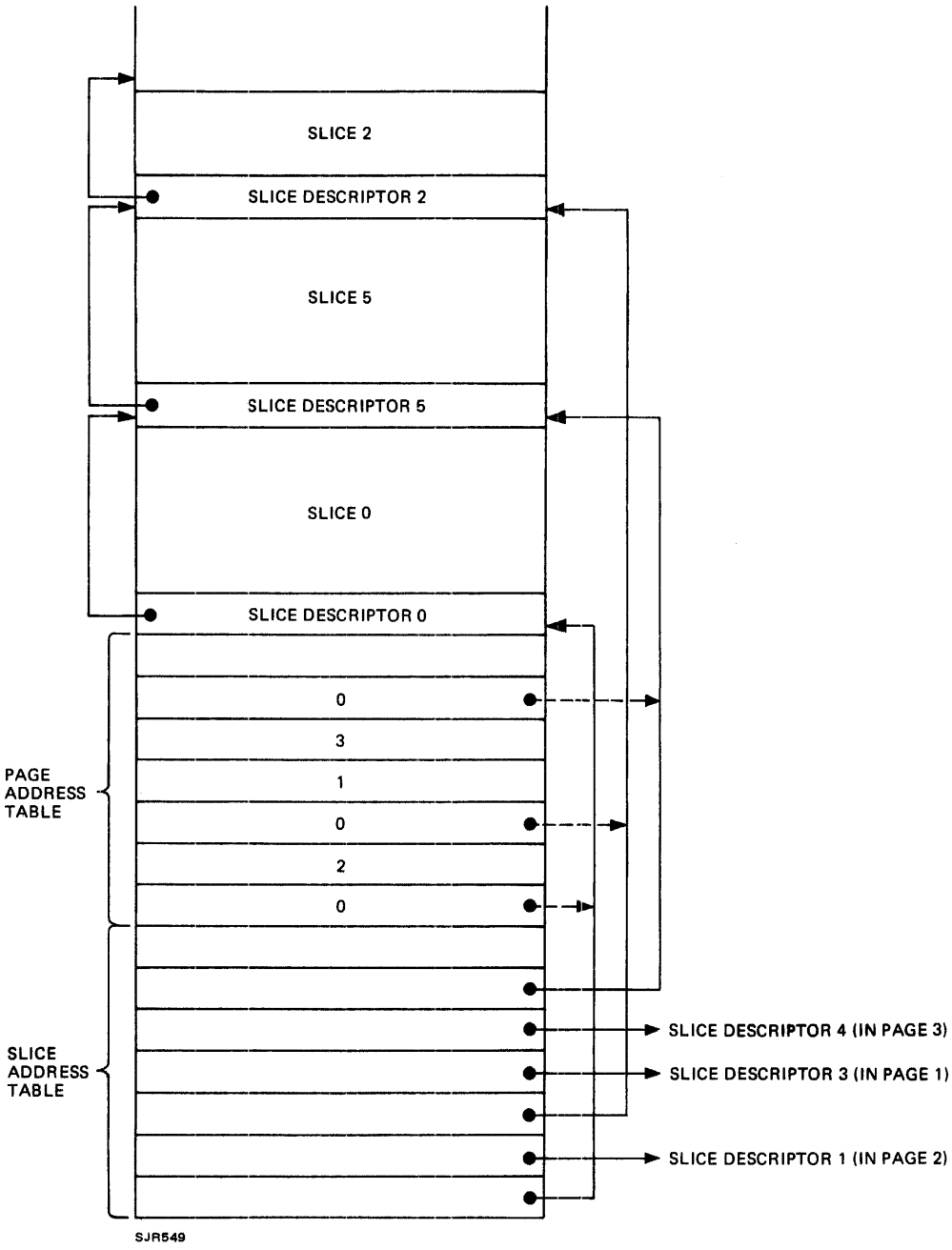


Figure 8-2. Page 0 Slice Memory

All Slices, whichever page they occupy, have an entry in the Page Address Table denoting that page, with a corresponding entry in the Slice Address Table giving the address of their Slice Descriptor.

The Page Address Table contains two bytes per entry; the first is the page number, the second a flag byte. Only the most significant bit is defined; if set, the slice is fixed in the page it is occupying and is not to be moved to another page by the MCP during memory management. Slice memory in non-zero pages begins with its first slice descriptor after a small Work Area; there is no Slice or Page Address Table.

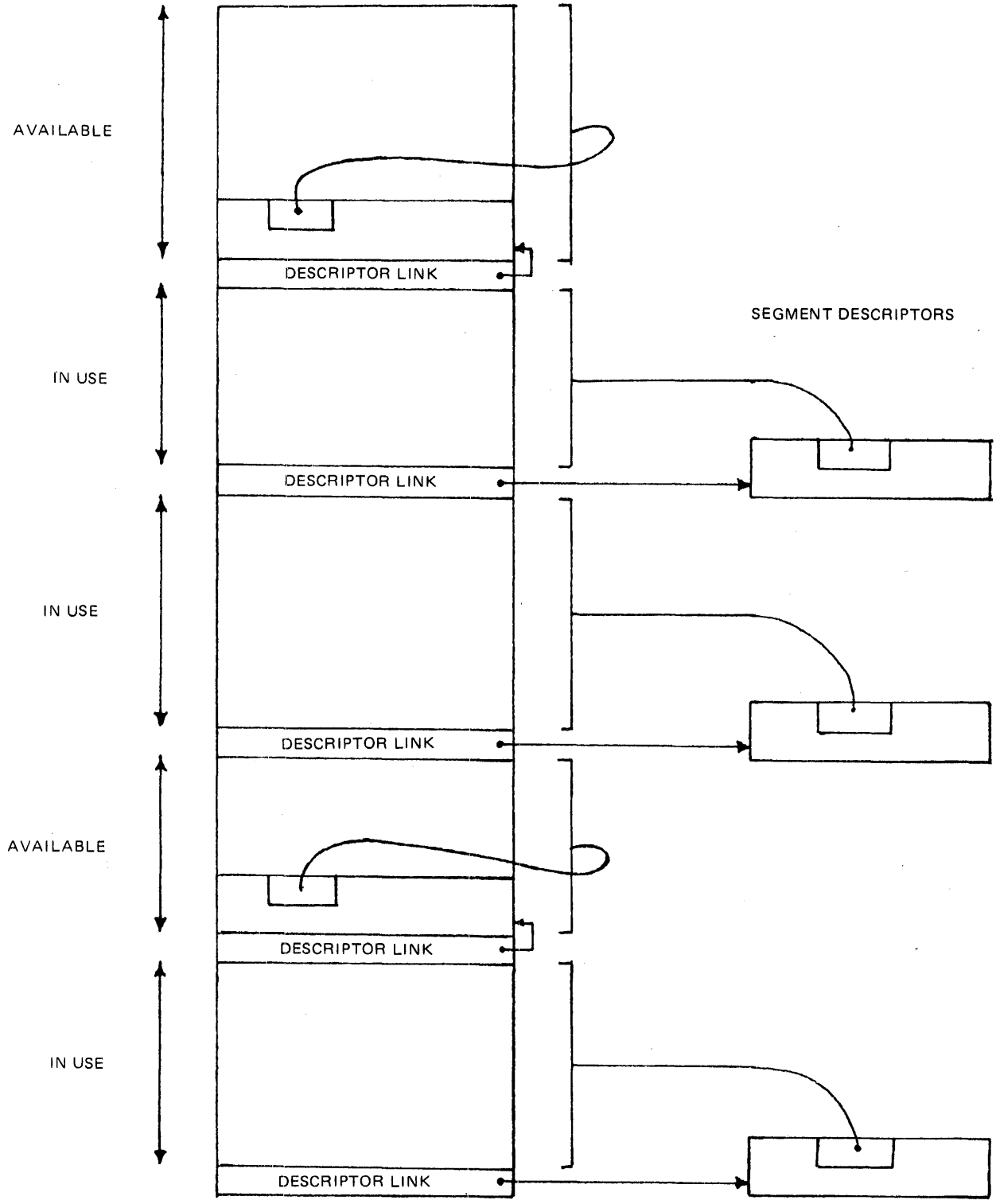


Figure 8-3. Overlay Arena

Each slice descriptor contains a field indicating where on a disk this slice resides. This address is used to fetch an absent slice into memory and to write an updated slice to disk if this becomes necessary. Also in the slice descriptor is a field indicating the length of the disk copy which is also used when fetching absent slices. Certain slices (see below) may be shared by multiple tasks and these slices have, in their slices descriptors, a count of the number of tasks using them. Whenever a task terminates, it decrements the user counts in all its slices and those slices with no users can be removed from memory.

Slices can contain any type of information: Data, S-code or micro-code. There are certain types of slices that have standard formats and special usage. These are described as follows:

The first 32 slices are reserved for use as task control blocks (TCB). Each task running in the system is assigned a slice as its TCB and the mix number of the task is the number of its TCB. The TCB contains all the information required to execute that task. It points to the interpreter and program segment tables. It contains the data segment tables and all interpreter and S-code work areas. Access to the TCB implies access to all parts of the run structure of a task. Certain system tasks are allocated TCB slices from the pool of 32. The mix numbers corresponding to these system tasks are never available for user tasks.

Any slice other than the first 32 can be a Program Control Block (PCB). Each task normally has associated with it a PCB which contains the program segment table and all other information associated with the program which remains static during the execution of the program. In COBOL, for example, the COP table (see COBOL S-machine), which remains unchanged by execution, is kept in the task's PCB.

The Interpreter Control Block (ICB) is analogous to the program control block in that it contains only static information. The ICB is used to gain access to the interpreter micro-code and other static interpreter information.

Both ICB's and PCB's have similar formats and are referred to by the collective name Code Control Block (CCB).

Another form of CCB is the MCP code control block which can hold a segment table for parts of the MCP. For example, OPEN/CLOSE is a code control block slice containing a segment table to the various routines required during the opening and closing of files. An example of a slice containing only data is a translation slice which holds information about the data translation to be performed for a particular peripheral.

SEGMENTS

In a program or interpreter, all code and data is maintained in segments. Whenever it is required to access code or data, a two part address is required; the first part is the segment number in which the required item is stored and the second part is the offset from the base of the segment of the required item. A normal task has three associated segment tables: The data segment table (DST) which is maintained in the TCB, the code segment table (CST) which is maintained in the PCB, and the interpreter segment table (IST) which is maintained in the ICB.

All segment tables are of identical format and consist of one entry, called the segment descriptor, for each segment associated with the table. Each segment descriptor consists of four fields: the length, address in memory and address on disk of the segment; and a set of flags indicating the current status of the segment. Segment numbers indicate the relative position in the segment table of the associated segment descriptor.

When accessing information in segments, the appropriate segment table is selected and the required segment number determines the required segment descriptor. The 'absence bit' in the descriptor flags

indicates whether the segment is present in memory or is currently on disk. If the segment is absent, indicated by the absence bit being set, then the virtual memory sub-system must be invoked to make present (in memory) the required segment. If the segment is present in memory, indicated by the absence bit being reset, then the memory address field points to the base of the segment. The required item's offset into the segment must be checked against the segment length fields to ensure that no attempt is made to access information outside the segment. Users of segments are expected to maintain two descriptor flags. The 'in-use' flag must be set whenever a segment is accessed. This flag is used by the virtual memory subsystem in deciding whether the segment should be returned to disk if room is required in the overlay arena. The 'update' flag should be set if the contents of the segment are altered, but before altering segment contents, the 'read-write' flag must be examined. If set, the read-write flag indicates that this segment can be written to. If reset, the flag indicates a 'read-only' segment which should not be altered. Whenever the virtual memory subsystem decides that a segment should be made absent from memory, only read-write segments with their update flags set need to be written back to disk since the disk copies of all other types of segment will still be up-to-date. Whenever the virtual memory subsystem makes a segment present in memory, it resets the in-use and update flags in the segment descriptor. Note that all code segments are read-only since they are shareable among concurrently executing tasks.

A segment can be marked as being locked by setting the lock flag in the segment descriptor. A request to the virtual memory subsystem to make present an absent, locked segment will result in the segment being added to the slice containing the segment descriptor. Once a segment is in slice memory it will not be made absent until all tasks using the slice have terminated or have been suspended to disk. Normally, the only user segments which are locked are data segments zero for MPL programs.

VIRTUAL MEMORY SUBSYSTEM

The algorithm used by the virtual memory subsystem in the swapping segment has already been described in Section 6. As described in that section, overlay code and data segments are kept in the overlay arena. The searches described in Section 6 are performed as a linear search of the overlay arena. The entire overlay arena consists of segments; available areas are maintained as segments with their own descriptors. Each segment, in the overlay arena, is preceded in memory by a pointer to its segment descriptor. This is illustrated in figure 8-3. The descriptor, described in this section, provides information about the status and length of the segment and by adding the segment length to its base address, the next descriptor pointer is found. The overlay arena, then, consists of a chain of segments which can be scanned by ascending memory address. For each available area a segment descriptor is built and stored inside the area. If the available area is not large enough to hold a descriptor, the area is filled with zeros, indicating a small available area.

TASK RUN STRUCTURE

For any program to be executed, the appropriate run structure must be built in main memory to allow the correct interaction of the program, interpreter and MCP. It is the job of the program loader to convert a program file described in Section 4 into the appropriate run structure, as illustrated in figure 8-4. The loader runs as a special task within the MCP with mix number 30. This task also handles the SCL processing and is therefore referred to as the SCL/LOADER task. When a task ends, its run structure must be dismantled and removed from memory. This job is performed by an MCP slice called TERMINATOR which runs as part of the terminating task.

TASK, INTERPRETER AND MCP INTERFACE

At the head of the TCB are fixed absolute pointers, maintained by the MCP, to other parts of the run structure. These pointers allow rapid accessing of key parts of the run structure and include pointers to the interpreter and program control blocks, and therefore to the interpreter and program code

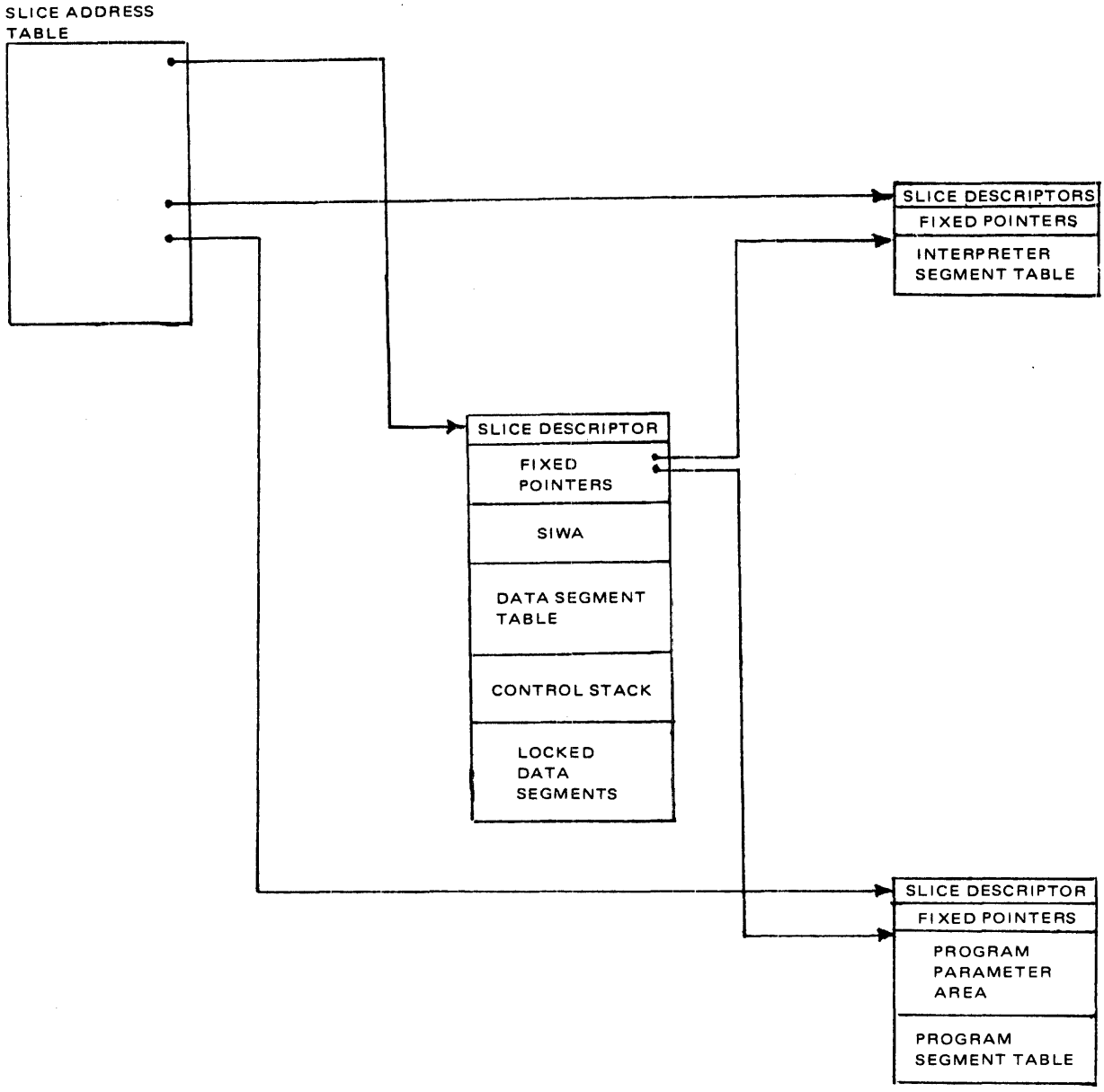


Figure 8-4. Program Run Structure

segment tables. The TCB is used to hold the S-interpreter Work Area (SIWA) which is used to store all the soft registers of a particular virtual machine. Beyond the SIWA is the data segment table which provides pointers to all data segments. Beyond this is the task control stack, used in communication with the MCP. Beyond this is a task-dependent register area for MCP usage. Any locked data segments used by the task will be placed in the TCB beyond the register area.

The SIWA format is dependent on the particular virtual machine.

Whenever the MCP passes control to a task, it expects to find restart information at the top of the task's control stack. This information is used to calculate the restart point for the task. It is the responsibility of each interpreter when yielding control to the MCP to ensure that appropriate restart information is left on the top of the stack. In addition, when the MCP is performing functions on behalf of the stack, it may use the space on the top of the stack as temporary work space or to call other parts of the MCP. The stacking and unstacking of entries is the responsibility of the calling functions. The virtual machine is free to make use of the stack space for its own internal use provided sufficient space is declared in the interpreter or program file (refer to Section 4) and providing the top of stack pointer (one of the fixed TCB pointers) is always maintained whenever the MCP is called.

Control of the processor can be yielded by a virtual machine to the MCP for one of three reasons:

1. The virtual machine requires the MCP to perform some function on its behalf.
2. The MCP performing a function for the virtual machine determines that it can cease execution awaiting a special event.
3. A soft interrupt has occurred.

In each case, a communicate is issued to the MCP and the stack should be loaded with the appropriate restart information. In case 1, the appropriate MCP function is called as a special subroutine of the virtual machine and may be regarded as a special S-op of the virtual machine. Case 2 is normally caused by a wait on an input-output operation to complete; it can also be caused by an attempt to gain a lock that is already in use. Case 3 is caused by the MCP determining that some event has occurred which changes a task's state from waiting to executable, either because of the completion of I/O or because a lock has become free.

It is the responsibility of the interpreters to check periodically the global soft interrupt flag, and to yield control to the MCP if it has been set.

To increase efficiency, the interpreter may calculate extra absolute pointers into the run structure in addition to those provided in the PCB. However, these pointers must be recalculated each time the interpreter is restarted since segments and slices may have moved in main memory due to the virtual memory activity.

EXECUTION PRIORITY ASSIGNMENT ROUTINE (EPAR)

The MCP maintains a number of tables in memory to enable it to pass control of the processor to the highest priority task able to use it. These tables are SCAN.TABLE, TIDT, ESCT and WAKT.

All four tables are 32 bytes long; one byte for each possible task. Entries in SCAN.TABLE and TIDT are arranged in task priority order, with corresponding entries referring to the same task. These two tables are arranged so that tasks with high priority are at the top of the table, and those with low priority are at the bottom; that is, the order from top to bottom is system tasks, class C tasks, class B tasks, class A tasks.

Each entry in SCAN.TABLE can take the value #80, #40 or #00, indicating that the corresponding task is executable with promoted priority, executable with normal priority or non-executable, respectively.

EPAR scans this table for an executable task and passes control to the first one it finds. The table is scanned first for an entry of #80. Failure to find one causes a second scan for #40.

Each entry in TIDT contains the mix number of the corresponding task, mix numbers being #00 – #1F.

ESCT and WAKT tables are arranged in ascending mix number order, #00 – #1F. The order of entries in these tables never changes. The entry in ESCT and WAKT for a particular task is found by adding that task's mix number to the base address of the particular table in question. Table 8-1 shows the meaning of the bit settings in an entry of ESCT.

Table 8-1. The ESCT Byte

Bits	Value	Meaning
7,6	00	Task can be run
	01	Task is short-waited
	10	Task is long-waited
	11	Task is swapped out
5	0	Task has promoted priority
	1	Task has normal priority
4,3,2,1	0000	Task is in an I/O wait
	0001	Task is in a zip wait
	0010	Task is in a timer wait
	0011	Task is in an AVR wait
	0100	Task waited on MCS Queue
	0101	Task waited on Data comm buffer
	0110	Task waited on MCS attachment
	0111	Task waited on station queue limit
	1000	Task waited on sub net queue limit
	1001	Task waited on MCS queue limit
	1010	Task waited on virtual memory I/O
0	0	Task is non-exchangeable; that is, class B or C
	1	Task is exchangeable; that is, class A

The priority bit, bit 5, is reset whenever a task gains a lock which is protecting a non-reentrant section of the MCP and is set whenever the task releases the lock. The entry in SCAN.TABLE for the task gaining and releasing the lock will change accordingly; that is, #80 while holding the lock, #40 after releasing it.

The exchangeable bit, bit 0, indicates that the entry for that task in both TIDT and SCAN.TABLE can be exchanged with the entries above or below it, provided that their exchangeable bit is also set. This provides a means within class A tasks of changing relative priorities so that I/O bound jobs are given a higher priority than compute-bound jobs. WAKT entries for executable tasks are #FF; for non-occupied mix slots #FD, and for non-executable tasks, WAKT indicates the reason for the task being waited. For certain wait conditions, bits 4 – 1 of the corresponding ESCT entry provide further information.

In the MCP idle state; that is, no task is executable, EPAR continually scans SCAN.TABLE. In this table, a hard interrupt will occur, the interrupt processor will make a task executable and EPAR will select that task for execution.

Time Slicing of Class C Tasks

A means of ensuring that each class C task in the mix is allocated a share of processor usage is provided. A count is maintained in each class C task's TCB. The count is decremented by the interpreter,

and on reaching zero, the interpreter yields control to the MCP which moves the task entry in both SCAN.TABLE and TIDT to the bottom of its class, resetting its count to the initial value. EPAR scan routine is then entered to select the next task for execution.

COMMUNICATES

Communicates are the means of communication between interpreters and the MCP. Most communicates are defined in Section 5. However, there is a class of communicates which only interpreters can use and which are machine dependent.

MASTER COMMUNICATE HANDLER (MCH)

The MCH is that part of the MCP which is called to handle communicates. It sets up temporary work space on the task stack, verifies that the communicate is valid and calls upon the appropriate routine to perform the requested function.

LOGICAL I/O HANDLING

When a file is opened, the OPEN communicate routine builds a file information block (FIB) and a file control block (FCB) from the declared file parameter block and the disk file header or file label, if these exist. The file information block and the file control block between them contain all the information required to perform logical I/O to and from the file. They contain the buffers, the I/O descriptions, record pointers, buffer pointers, file area address, and so on, associated with that file. The FIB contains task-dependent information. The FCB contains task-independent information and is held as a separate entity in a shared-file slice if the file is opened for SHARED use.

MACHINE DEPENDENT COMMUNICATES

There are two machine dependent communicates defined for the B 90 MCP:

Verb	Verb Value	Parameter	Function
YIELD	70	0000	Yield control to the MCP, enter EPAR leaving the calling task in its current state
GETSEG	71	Absolute memory address and page of the required segment descriptor	Request to make the required segment present in memory
PUTSEG	72	As GETSEG	To make specified segment absent from memory
SUSPEND	74	None	Stop the task with a 'STOPPED' message
LOCAL.LANG	75		Local language communicate
LINK FIBS	76	In CPA: Disk FIB seg # Tape FIB seg # Speed (0 = > run slowly, 1 = > try to run fast)	To link disk and tape FIBs for fast LD

In all communicates, the interpreter must ensure that correct restart information is left on the top of the task's control stack in order that the task can be restarted at the correct point.

PHYSICAL INPUT-OUTPUT

All physical input and output operations are performed under interrupt control and completely asynchronously from the rest of the system. Each peripheral has associated with it a Peripheral Handling

Table (PHT) and a device dependent routine (DDR). Whenever an interrupt occurs, the Master Interrupt Processor (MIP) is invoked. It scans all peripherals in order of priority to determine which device to service. Figure 8-5 is a simplified flow chart of physical input-output processing. No interrupt processing will disturb any memory location not directly involved in the operation and the processor registers are restored to their original values when all requesting devices have been serviced. When a buffer transfer has been completed and the task which requested the transfer is waiting on an input-output event, then the task is set executable in the ESCT and the global soft interrupt flag is set by setting it to binary zero. The distinction between hard and soft interrupts ensures that processor time is not wasted in saving, restoring and switching tasks every time a peripheral device requires attention. Only when a significant event takes place which changes the state of another task will the MCP be invoked to determine whether the current task should continue or whether a higher priority task can now run.

MASTER INTERRUPT PROCESSOR (MIP)

The MIP is invoked for each hard interrupt. It immediately saves all processor working registers and scans the I/O channels in peripheral priority order. When a requesting device is found, its peripheral handling table is examined and unless some special event is flagged either in the status from the device or by the PHT status, the next "page" of characters is transferred between the current buffer and the device, where a 'page' is usually the number of characters that particular device can accept or transmit at a time.

If some special event is flagged, the appropriate DDR is invoked to resolve the problem. When a device has been serviced, the peripheral scan is re-initiated to determine whether any other device is requesting. Only when this scan finds no requesting device are the processor registers restored and control of the processor returned to the interrupted task.

The MCP also includes the routines for handling the I/O descriptor queues attached to each PHT. Each task requesting a physical I/O transfer must supply a descriptor describing the required transfer. The descriptor provides information as to the unit through which the transfer is to take place, the length of the transfer, the address, in main memory, of the buffer and flags for status information to be passed between the task and the I/O handler. This descriptor is linked to the appropriate PHT queue and the task whose descriptor it is,must not access or change information in the buffer or descriptor until the flag indicating transfer complete is set. When the transfer for the top descriptor is completed, the descriptor is removed from the queue and the transfer described by the next descriptor is initiated. The complete descriptor is marked as being complete and if the task which initiated the transfer is waiting on an I/O event, then it is marked as being able to be run in the ESCT.

When a queue is exhausted, the device is 'shut down' and should not interrupt until the next descriptor is queued and initiated.

Interrupt processing must proceed at a brisk rate since it is impossible to service another task until the current one has been completed. For this reason some low priority interrupts are not serviced as part of the MIP but are passed on to the AVRIL task for handling. These interrupts include a device being readied, or powered down; for example, a disk or cassette. In such cases, tables must be updated to reflect the change in status. In the case of a disk or cassette being made ready, I/O transfers must be performed in order to read the label. In these cases, AVRIL is set as runnable, special flags are set to indicate the reason and the global soft interrupt flag is set. AVRIL, being a higher priority task than any user task, will perform the required processing as soon as the currently running task yields to the MCP.

MCP DIAGNOSTICS

The MCP trace feature provides a selective functional trace of virtual machine functions.

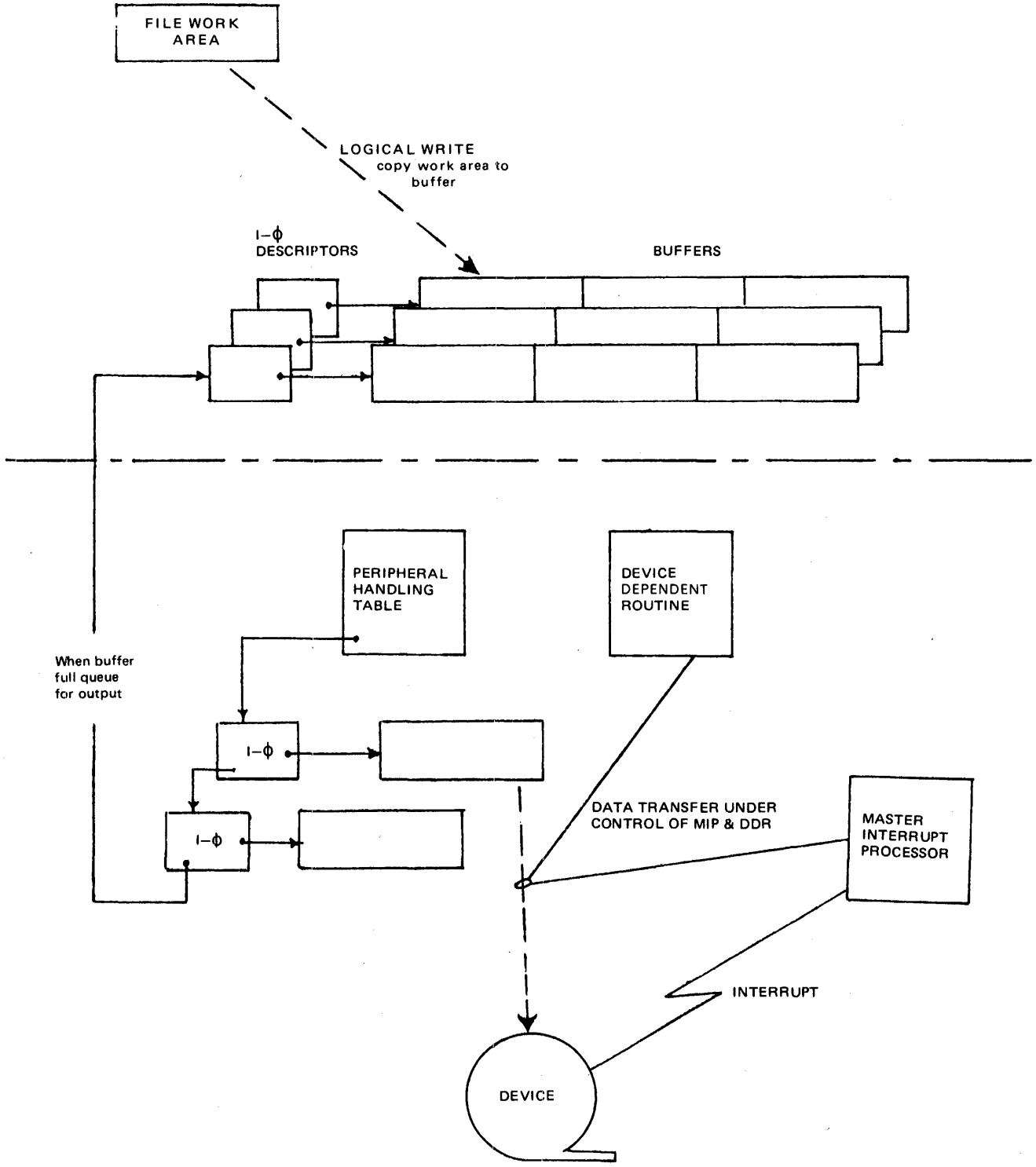


Figure 8-5. I/O Processing

Diagnostics are grouped into 16 classes distinguished by a hexadecimal digit 0-F. Each facet of the virtual machine uses a different diagnostic class, as follows:

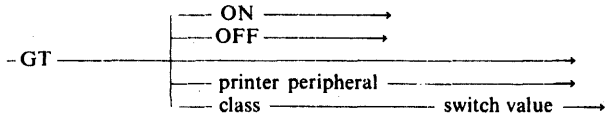
Diagnostic Class	User
0	Open/Close
1	Indexed file handling
2	Accept/Display
3-7	Utilities etc.
8	Automatic Volume Recognition
9	Disk Clean and slice handling
A	Allocate/Deallocate
B	Interpreters
C	Master Communicate Handler
D	Virtual Memory
E	Task Control
F	I/O Q-handler

Within each diagnostic class there are two types of diagnostic – register diagnostics and memory diagnostics. Register diagnostics print the contents of the machine registers at selected points in the code. Memory dignostics print potentially interesting areas of memory at selected points in the code.

Each diagnostic in the code has an associated severity index expressed as a hexadecimal digit 0-F. Corresponding to each class and type of diagnostic is a diagnostic switch. When a diagnostic is encountered in the code, the appropriate printing action takes place if the severity index of the diagnostic is greater than or equal to the value stored in the corresponding switch.

The diagnostic switches are stored in an array of 16 one byte cells indexed by class. Each cell contains a two digit hexadecimal number, the more significant digit being the register diagnostic and the less significant digit being the memory diagnostic for that class.

Control of the trace facility is by means of the GT command, the syntax of which is shown below:



GT ON and GT OFF respectively enable and disable the diagnostic printing. Normally the diagnostic printing is disabled and will be set off by a warmstart. The printer peripheral option allows the redirection of diagnostic output to the named printer. Diagnostics are directed to SPA, if available, after each warmstart. The class switch value option enables the setting of the diagnostic switches. Class is a single hexadecimal digit indicating the required switch and switch value is a two digit value to be stored. If no options are specified, GT prints the 16 diagnostic switches.

If diagnostics are enabled, the lights above PK 17 to 24 continuously display the current diagnostic trace point.

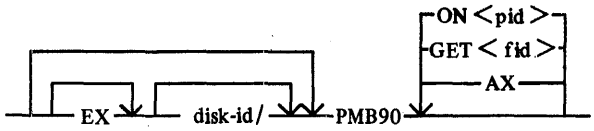
MEMORY DUMP ANALYZER – PMB90

This utility will produce an analyzed listing on a line printer (or console printer if no line printer is available) of the contents of a memory dump cassette tape or disk file. A memory dump is produced

by the Warm Start memory dump facility described in the CMS Software Operation Guide, Form number 2015228). A tape or cassette has the label "MEMDUMP/MEMORY". A disk file has the name "MEMDUMP".

The utility consists of the object program "PMB90" and two data files identified "PMBM.XXXXX" and "PMBO.XXXXX" where XXXXX corresponds to release level of MCP in use at the time of taking the memory dump. These files must be on the system disk when running PMB90.

The initiation syntax is as follows:



PMB90 looks for a cassette or tape. If it cannot find one, it looks for a disk file. If AX is specified, the anaylzer communicates with the operator via accepts and displays on the system journal, otherwise the console will be used for communication. If ON is specified, MGMDUMP is assumed to be on disk 'pid'. If GET is specified, the disk file is assumed to have the name 'fid'.

The analysis takes place in an interactive manner. The analyzer prints only those parts of the memory structure specified by the user.

In the event of difficulties, a help function is implemented which provides instructions for the use of the analyzer. The help function is invoked by specifying "HELP" when the analyzer prints a prompt.

A complete dump analysis can be produced by specifying "PRINT ALL.MEMORY" after a prompt. This will produce a complete analysis of all parts of memory.

SECTION 10
B 1800 DEPENDENT IMPLEMENTATION

MCP STRUCTURE

INTRODUCTION

The B 1800 CMS Master Control Program (MCP) is composed of various modules each with its own functional task. Within the MCP itself the various functional modules interface with each other through a co-routine structure. The main functional modules are:

GLOBAL
LOGICAL I/O
PHYSICAL I/O
LOADER
TERMINATOR
MEMORY MANAGER
ODT/SCL HANDLER

Each module consists of a memory resident segment and a number of overlayable segments. The MCP itself is designed to run in a minimum of 50K bytes of memory.

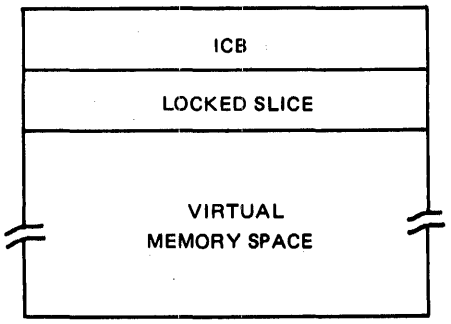
Two mix slots are reserved for MCP functions:

MIX #0	is reserved for the sub-task of the ODT/SCL HANDLER Manager which handles the servicing of the ODT.
MIX #15	is reserved for the sub-task of the MCP which handles the overlaying of MCP segments.

CO-ROUTINE STRUCTURE

Co-Routine Concept

The co-routine structure is implemented as follows. The run structure of each user job (user program, compiler, utility) consists of a data structure composed of the Interface Control Block (ICB), the Locked Slice and a Virtual Memory area for data and code segments:



All three data areas are contiguous and always remain so throughout the life of the job (except in the special case where a job is rolled-out, in which case only the ICB remains in memory). The total

memory space assigned to a job is referred to as the job's 'partition'. The address of each user job partition is maintained in the Memory Assignment Table (MAT) and this is also the address of that job's ICB. For the purposes of this discussion the contents of the job's partition can be considered as data which is used primarily by the appropriate language interpreter to execute the job steps desired. When the interpreter cannot perform the desired function, control of the partition is passed to the appropriate MCP module for the purpose of executing that function (loading a segment, or actioning an I/O operation). In conceptual terms, the execution of the job can be viewed as the performance of a variety of tasks, an interpreter and a number of MCP tasks. The interface between the various tasks is effected through, and controlled by means of, the ICB. The task currently executing for any particular job then controls the job's partition and therefore its ICB.

Co-Routine Implementation

A level number is assigned to each main task; the lower the level number, then the closer that task is to the job itself. The assignment of levels is shown in table 10-1.

Table 10-1. MCP Functional Levels

Level	Function
0	INTERPRETER
1	LOGICAL I/O
2	MEMORY MANAGEMENT
4	PHYSICAL I/O
6	CONTROL
7	OPERATOR INTERFACE MANAGEMENT

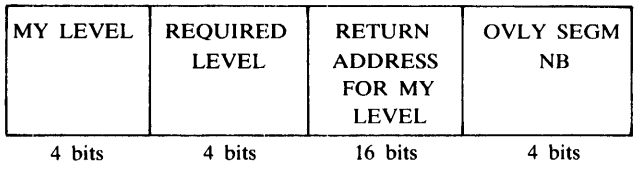
In general terms control passes from a low to a high level to request the performance of a particular task, and control returns from a high to a low level as each task is completed. So, for example, a communicate requesting the read of a particular record might follow the following route:

Level	Action
0	INTERPRETER decodes the communicate and passes control to LOGICAL I/O
1	LOGICAL I/O analyzes the communicate and has to request a PHYSICAL I/O to get the buffer containing the requested record
4	PHYSICAL I/O initiates the I/O and when complete returns to LOGICAL I/O
1	LOGICAL I/O now has the requested record in the buffer but the work area data segment is not in memory and so it must request MEMORY MANAGEMENT to load the segment
2	MEMORY MANAGEMENT checks whether sufficient memory space is available Assuming this is so, it requests from PHYSICAL I/O to read the segment from the Virtual file
4	PHYSICAL I/O reads the data segment into memory and returns to MEMORY MANAGEMENT
2	MEMORY MANAGEMENT checks if the segment was loaded successfully (no Read error) and passes control back to LOGICAL I/O
1	LOGICAL I/O transfers the requested record from the buffer to the work area and returns to INTERPRETER
0	INTERPRETER continues with the next S-OP

The two levels not shown in table 10-1 are used internally by the MCP for its own tasks:

Level	Function
3	OVERLAY HANDLER FOR MCP
5	SWAP (ASYNCHRONOUS BUFFERING)

The mechanism used to control switching from a task to another task is ICB.COMMUNICATE.ROUTING which has the following format:



These fields are set up by a task when it requires a function to be performed by a task with a higher level number.

For example, if an interpreter detects a communicate S-OP it will set the ICB.COMMUNICATE.ROUTING as follows:

0	1	01A8	0

where @01A80@ is the address relative to the base address of the interpreter to which control should return after completion of the communicate.

NOTE

As a B 1800 micro address is always modulo 16, the return address is stored divided by 16 and the correct address generated when the return is effected.

The OVLY SEGM NB is set to zero when a task is switching directy to another task (SEGM NB 0 is always resident) or when return is to the resident section.

However, when a return is desired to an overlayable segment, then the OVLY SEGM NB is set to the segment requested. In this case the return address is relative to the base of the overlay.

For example:

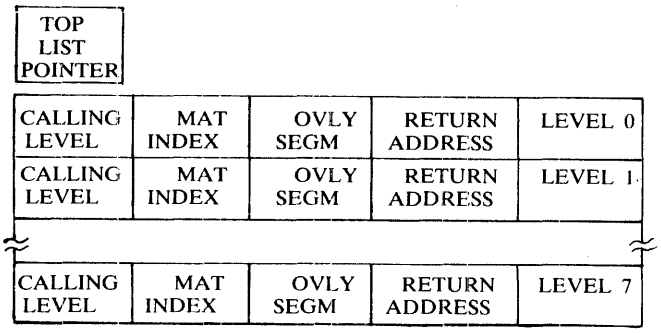
1	4	01B3	2

is a call to PHYSICAL I/O with a return to SEGM NB 2 of LOGICAL I/O at a relative address of @1B30@.

The actual switching is done by the routine COMMUNICATE.SWITCH to which the task branches and which itself branches to the requested task.

In the case of a task wishing to transfer control from its resident section to one of its overlay segments (OVLY SEGM NB > 0) then it will set REQUIRED LEVEL to 3 (OVERLAY HANDLER), RETURN ADDRESS to 0 and OVLY SEGM NB to the required SEGM.NB. It then switches to the OVERLAY HANDLER which checks whether the segment is already in memory, in which case it will branch back directly to the start of the Overlay. If it is not in memory, then the MX for the partition will be set to WAITING OVERLAY and MIX #15 of the OVERLAY HANDLER is primed to load it. On completion of the load, control is then passed to the segment.

The mechanism used for ensuring that control returns correctly is the ICB.CO-ROUTINE.LIST in the ICB. Its format is:



The COMMUNICATE.SWITCH stores the field MY.LEVEL in TOP.LIST.POINTER and the OVLY SEGM NB and Return Address in the CO.ROUTINE list at MY.LEVEL. MY.LEVEL is also stored in CALLING.LEVEL of REQUIRED.LEVEL.

Using the first example above:

(MY.LEVEL = 0, REQUIRED.LEVEL = 1,
RETURN.ADDRESS = @1A8@)

TOP.LIST.POINTER will be replaced by 0, the RETURN.ADDRESS at level 0 by @1A8@ and CALLING.LEVEL at level 1 by 0. The return route is always through the TASK.SCHEDULER in GLOBAL. The level in TOP.LIST.POINTER is used to index into the CO.ROUTINE list. The OVLY SEGM NB is examined. If it is zero then the MAT index is used to obtain the absolute base address of the resident section. By adding the RETURN.ADDRESS to the base, the address to which control is returned is obtained. If the OVLY SEGM NB is not zero, then the Segment Table for MCP overlays is examined. If the segment is in memory its base address is obtained and control returned at the relative address in it. If not, then the segment is loaded using MIX #15 which means control is transferred when the MIX is next selected after segment load has completed.

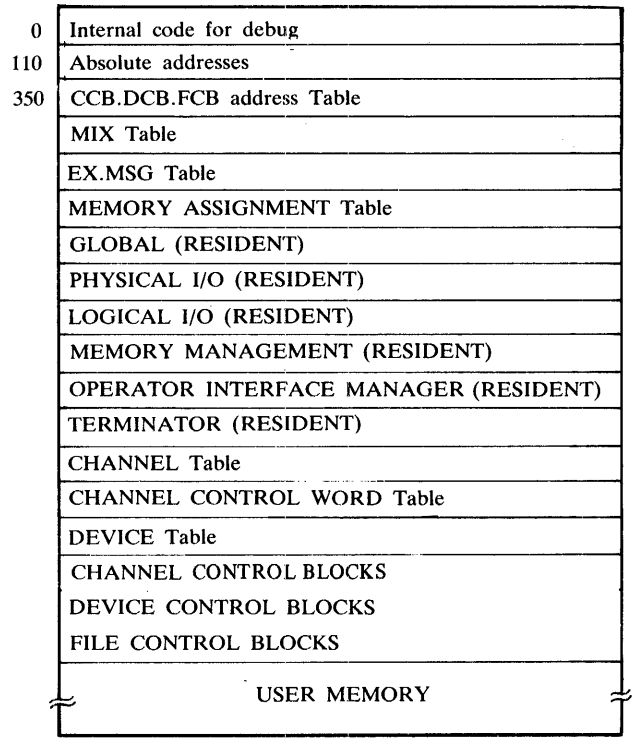
Restart Mechanism

In certain cases an MCP module may not be able to fulfil a request immediately because it is itself waiting for some prior action to be completed. For example, if a block of data is in the process of being read from disk, then a request to write into that block cannot be honored until the read has completed.

In situations like this, the called task returns a special fetch value (@FFFFFF@), which is defined to mean 'restart the operation again', to the calling task. The calling task whenever it is re-scheduled will attempt the same communicate repeatedly until either a fatal or successful indication is returned.

MEMORY STRUCTURE OF MCP

At System Initialization time, the beginning of memory is structured as follows:



NOTE

The above addresses are in hexadecimal and represent a bit displacement.

Absolute Addresses

Each address occupies four bytes in memory in the hexadecimal form @93AAAAAA@ where @AAAAAA@ is an absolute bit address. These addresses are in the following order:

MAIN.SEG.BASE
MAIN.IDLE.LOOP
COMMUNICATE.SWITCH
MIX.TABLE
MEMORY.ASSIGNMENT.TABLE
EX.MSG.TABLE
BOOTSTRAP (in GLOBAL)
INTERRUPT.REGISTER
INTERPRETER.TABLE
CHANNEL.TABLE
CCW.TABLE
DEVICE.TABLE
MOD.ID

TIMER.WORDS
OS.WORK.AREA
PHIO.BASE
OS.ICB
VM.ADDRESS

CCB.DCB.FCB Address Table

Contains the start and end addresses of Channel Control Blocks, Device Control Blocks and File Control Blocks for Serial Devices, Magnetic Tapes and Disks.

MIX Table

Contains one entry per MIX #, each entry giving the program name and information about the priorities, the virtual memory in use, the program status and the ZIP variants (if MIX # was zipped).

EX.MSG.Table

Contains the information pertaining to a load request, the pack-id, and program name, the memory size requested, various flags, and, if applicable, the initiating message address and length.

MEMORY.ASSIGNMENT Table

Contains 29 entries:

16 entries for user jobs
4 entries for language interpreters
9 entries for main MCP tasks

Each entry contains the start address of a partition, the length of a partition and flags used by Memory Management.

CHANNEL.Table

Contains, for each channel, the control ID (as returned by a TEST STATUS command), the Mnemonic (DK, DP, MT) and the Device-type (Serial, Tape, Disk) of the device connected.

DEVICE Table

Contains for each kind of device all the information required to construct the CHANNEL Table.

Memory Management

Contains at the beginning of its partition, internal tables which are:

MM.MEMORY.MAP with one entry per partition containing status flags.
VM.JOB.TABLE with one entry per MIX reflecting current or requested VM action.
MM.JOB.TABLE with one entry per partition, giving all the information about any communicates rerouted from VM to MM.
MM.WAKE.UP.TABLE containing all the information for ROLL-IN/ROLL-OUT processing.

GLOBAL

The GLOBAL module is functionally composed of four independent sub-modules:

1. Main idle loop to which all modules return when they have nothing more to do.
2. COMMUNICATE.SWITCH which is used when modules want to branch directly to other modules.
3. BOOTSTRAP which is the initial phase of the actual load process for user jobs.
4. CLAC which handles all Class C Communicates (ACCEPT/ZIP/DISPLAY). This is the CONTROL level (level #6).

MAIN IDLE LOOP

The main idle loop performs the following actions in the order specified:

If an interrupt is detected (CC bit 2 set), then it branches to the Handle-Service-Request routine in PHYSICAL I/O.

If a timer interrupt is set, updates internal timer registers (the MCP maintains a clock with a resolution of 1 second).

If a Result Descriptor is pending, then it branches directly to the Result Descriptor Examination routine in PHYSICAL I/O.

Every 800 milliseconds, it branches to the Test Peripheral routine to test whether any status change has occurred on any channel.

Every second, it branches to the WAKE.UP routine in the Memory Manager.

If necessary, it branches to the MM routine in the Memory Manager, or immediately enters the Scheduler.

When the Memory Manager returns control, the Scheduler is entered.

The Scheduler itself is divided into two parts.

The first part of the JOB.SCHEDULER determines which job, if any, will acquire the processor. Two factors are taken into account when deciding whether a job will or will not be scheduled. The first factor is the status of the job as marked in the PROGRAM.STATUS in the MIX.TABLE. Only those jobs which are not waiting on some internal or external event (I/O complete, not waiting memory management, operator input, duplicate file) are candidates for selection. However, because of the asynchronous nature of the buffer management (refer to LOGICAL I/O), a job may not itself be waiting for an I/O but an I/O has completed for a file within it. Therefore, if an asynchronous physical I/O operation has completed, then control is passed first to LOGICAL I/O to handle the I/O complete. The second factor is the priority of the job. Two fields in the MIX.TABLE control the priorities; INITIAL .PRIORITY and INSTANT.PRIORITY

INITIAL.PRIORITY is set to a value of 2 for CLASS A jobs, 4 for CLASS B jobs, and 6 for CLASS C jobs by the LOADER, and initially the INSTANT.PRIORITY is set to the same value. However, whenever a job satisfies all criteria for scheduling, its INSTANT.PRIORITY is incremented by 1. Whenever more than one job is available for scheduling the job ultimately selected is the job with the value of INSTANT.PRIORITY. Note that if several jobs are encountered with the same highest value of INSTANT.PRIORITY, the first one met is scheduled. Once a job has been scheduled, its INSTANT.PRIORITY is reset to the value of its INITIAL.PRIORITY.

The second part of the Scheduler is the TASK.SCHEDULER which transfers control to the task (interpreter, MCP function) which is required for the job. Control is transferred using the information contained in the ICB.CO-ROUTINE list, as outlined in the Co-routine Implementation paragraph.

The MIX.TABLE is always scanned from MIX #0 to MIX #15. Note that this means that the ODT/SCL HANDLER (MIX #0) and the MCP OVERLAY HANDLER (MIX #15) are candidates for selection each time the Scheduler is entered.

COMMUNICATE SWITCH

The COMMUNICATE.SWITCH routine updates the appropriate entry in the CO-ROUTINE list, updates the TOP.LIST.POINTER and branches to the required module.

BOOTSTRAP

The first action of this routine is to find a MIX slot that is not currently in use. If a free slot cannot be found, a message '[052] MIX FULL' is issued and the load process terminates. The scan of the MIX table is always from MIX #1 to MIX #14 and the slot allocated is always the first one found (in other words MIX #s are not cyclic). The next phase is to allocate the OS.ICB (in GLOBAL) to load the LOADER. This is done by placing the address of the OS.ICB in the Memory Assignment Table at the position corresponding to the allocated MIX #. To enable a direct switch to the second phase of the bootstrap process which involves loading the LOADER module from disk, level 0 (the interpreter level) of the OS.ICB is set to the index of PHYSICAL I/O in the MAT and the return address to the address of the PHYSICAL.BOOTSTRAP in PHYSICAL.I/O. This ensures that when the ICB is next selected by the JOB.SCHEDULER, transfer is passed automatically to PHYSICAL I/O. The final action of this phase is to suspend MCP.SCL by setting MX.BOOT.RUNNING. This effectively prevents input of any further execute messages by the operator (and incidentally output of MCP or task-initiatied displays) while PHYSICAL.BOOTSTRAP is running. This is done because PHYSICAL.BOOTSTRAP routine is not re-entrant. Finally, an exit is made to the PHYSICAL.BOOTSTRAP in PHYSICAL I/O by way of the main idle loop in GLOBAL.

CLASS C COMMUNICATES

These communicates (ZIP/DISP/PAUSE/ACCEPT/WAIT) are rerouted to GLOBAL by LOGICAL I/O. Some of these communicates (DATE/TIME,WAIT,ACCEPT) require only that MCP tables are accessed and updated if necessary. Others (ZIP/PAUSE/DISPLAY) are initially checked and passed to the Operator Interface Manager for action.

Also handled by this routine are certain internal communicates (DS/DP).

LOGICAL I/O

INTRODUCTION

The function of LOGICAL I/O is to service all legal requests for an I/O either by transferring data between file buffers and user work areas or by setting up the parameters for a physical I/O operation, when data must be physically transferred to or from file buffers, and issuing the necessary communicate to PHYSICAL I/O. Therefore, LOGICAL I/O has as its main sub-tasks, analysis and validation of all communicates received, management of file buffers and interfacing with PHYSICAL I/O. Inherent in the design of the B 1800 CMS MCP is the independence of LOGICAL I/O from any knowledge of actual disk addresses or any media-related accessing features.

Analysis and Validation of Communicates

LOGICAL I/O examines the Communicate Parameter Area (CPA) which is contained within the ICB. The communicate is checked for validity, both in terms of adherence to the syntax for that communicate and for consistency and application within its environment. If the communicate is incorrect, the appropriate fetch value is returned to the requestor.

When the validity of the communicate has been established, LOGICAL I/O enters the next main subtask.

File Buffer Handling

Communicates fall into two main categories; File-oriented (OPEN, CLOSE) and Record-oriented (READ, WRITE, REWRITE, DELETE). File-oriented communicates require the building (OPEN) or dismantling (CLOSE) of a FILE INFORMATION BLOCK (FIB) which contains the file buffers and status information about the file and which is used as the interface between LOGICAL I/O and PHYSICAL I/O. Record-oriented communicates involve the transfer of data between the file buffers in the FIB and the user's work area and as such can be actioned immediately by LOGICAL I/O if the requested record exists in any of the file buffers. If the record does not exist, then it is necessary for LOGICAL I/O to issue a communicate to PHYSICAL I/O to fill a buffer (READ-related) or empty a buffer (WRITE-type) or both (if the file is an Input-Output file). LOGICAL I/O attempts to fill or empty buffers asynchronously of requests to access any particular buffer.

Interface with Physical I/O

There are three situations in which LOGICAL I/O needs to issue a communicate to PHYSICAL I/O:

1. When a request cannot be fulfilled until some prior physical I/O operation(s) have been completed.

   For example:

   a. OPEN/CLOSE of a disk file requires PHYSICAL I/O to access and update disk structures (Disk Field Headers, Available Table) to copy certain information to/from the FIB.
   b. READ is requested for a record which is not in any buffer and which therefore requires a physical I/O operation.

2. When LOGICAL I/O has to fill/empty a buffer asynchronously; that is, not as a result of a user I/O request (LOGICAL I/O will attempt to do this after a successful data transfer and after completion of a physical I/O operation).

3. When a request for a READ/WRITE necessitates access to a buffer which is in the process of being filled or emptied as the result of an asynchronous I/O operation.

In case (1) the communicate issued by LOGICAL I/O indicates the required action and an instruction to WAIT until the I/O operation is complete (this is achieved by setting the bit MX.WAITING in the MIX Table for the job).

In case (2) the communicate issued by LOGICAL I/O will indicate the required action only. Therefore, execution of the job continues normally except that when the I/O operation is complete LOGICAL I/O will acquire control for only that amount of time needed to analyze the result. This particular case utilizes the SWAP level (level #5) mechanism as follows:

Level	Action
0	Interpreter passes control to LOGICAL I/O to handle an I/O communicate (READ).
1	LOGICAL I/O transfers data immediately. It can also fill a buffer. Therefore it issues a READ but sets MY.LEVEL in ICB.COMMUNICATE.ROUTING to level #0 so that when PHYSICAL I/O has initiated the I/O the Scheduler will switch back to level #0.
0	Interpreter continues execution.

However, at some finite time, two events will have occurred:

1. The I/O will complete.
2. The job will at some stage yield the processor, and the main idle loop will be entered.

The first time that the Scheduler is entered after event (1) is when it has recognized that an asynchronous I/O has completed so the TOP.LIST.POINTER of the job will be set to level #5 (the SWAP level) and the calling level of level #5 will be set to the original TOP.LIST.POINTER. Level #5 is in fact the point to which all communicates issued by LOGICAL I/O return. Control then passes to LOGICAL I/O to handle the examination of the Result Descriptor and updating of pointers. When the exit from the LOGICAL I/O to the main idle loop occurs, then the task scheduled prior to the SWAP operation is restored as the next task in TOP.LIST.POINTER.

In case (3) LOGICAL I/O issues a WAIT communicate only and sets the RESTART fetch value in the requestor's ICB. On receipt of the WAIT communicate PHYSICAL I/O sets the job's MIX status as WAITING. In this manner the job is now waiting for the completion of what started out as an asynchronous I/O.

All communicates issued by LOGICAL I/O return to the same point for updating buffer status and pointers and determination of the success or failure of the initial communicate to LOGICAL I/O.

MANAGEMENT STRATEGY

Each file is given the number of buffers declared in its FPB except in the case of a file opened with ACCESSMODE = RANDOM, in which case the number of buffers always defaults to 1. Associated with each buffer is a byte which indicates the current status of the buffer:

Status	Meaning
FREE	The buffer is available for data transfer from PHYSICAL I/O (INPUT) or from a user work area (OUTPUT).
BUSY	The buffer is currently in the process of being filled/emptied by PHYSICAL I/O.
TRANSFERABLE	Valid for INPUT only. The buffer contains data which is available for transfer to a user work area.
TO.BE.REWRITTEN	Valid for I/O only. Data in the buffer has been changed and therefore the buffer must be written back.
WRITABLE	OUTPUT only. The buffer is available for transfer to PHYSICAL I/O
ERROR	A PHYSICAL I/O operation on this buffer was unsuccessful (parity error, timeout), therefore its contents are not usable.

The buffers allocated to a file are handled as a cyclic chain, so buffer pointers are incremented by 1 modulo the number of buffers. For example, if a file has three buffers, then buffer #1 is always the next buffer after buffer #3. Three buffer pointers are maintained by LOGICAL I/O.

LOG.BUF.INDEX	The BUFFER # of the current buffer
PH.BUF.INDEX	The BUFFER # into which PHYSICAL I/O will transfer data on receipt of the next READ from LOGICAL I/O

OUT.PH.BUF.INDEX — The BUFFER # from which PHYSICAL I/O will transfer data on receipt of the next WRITE

Associated with the LOG.BUF.INDEX is a field LOG.BLK.NB which contains the block # or the current buffer within the file. From this value are calculated block #s which are passed to PHYSICAL I/O when buffers are filled/emptied. Block # is zero-relative and is converted by PHYSICAL I/O into absolute addresses (for disk) or physical action (non-disk device). Therefore, LOGICAL I/O is concerned only with relative locations within a file and need not concern itself at all with the characteristics of the physical container of the file data.

The following diagram illustrates the buffer handling for a file with MYUSE = I/O, ACCESSMODE = SEQUENTIAL, DEVICE = DISK, ORGANIZATION = SEQUENTIAL and NO.BUFFERS = 5.

		STATUS	BLK #
	BUFFER # 1	BUSY	228
OUT.PH.BUF.INDEX	BUFFER # 2	TBR	229
LOG.BUF.INDEX	BUFFER # 3	TFR	230
	BUFFER # 4	TFR	231
PH.BUF.INDEX	BUFFER # 5	FREE	NA

BUFFER #1 is currently in the process of being transferred to disk. BUFFER #2 is marked 'TO.BE.-REWRITTEN'. BUFFER #3 (which is Block #230) is the current buffer from which communicates from the interpreter will be filled. BUFFER #4 is marked as 'TRANSFERABLE' (it contains valid data and is available for processing) and BUFFER #5 is free. When the I/O for BUFFER #1 completes, its status wil revert to 'FREE' and BUFFER #2 will be transferred to disk asynchronously, thereby changing its status to 'BUSY'. OUT.PH.BUF.INDEX is incremented by 1 and now points to BUFFER #3. Communicates are filled from BUFFER #3 until a communicate is received which accesses Block #231. At that time LOG.BUF.INDEX is incremented by 1 and now points to BUFFER #4. The status of buffer #3 is changed to 'TO.BE.REWRITTEN', if any record within Block #230 was rewritten, or to 'FREE' if no rewrites were actioned. When BUFFER #2 completes, its status changes to 'FREE' and a read is issued to fill BUFFER #5 (with Block #232), thereby causing its status to change to 'BUSY'. PH.BUF.INDEX cycles round to point to BUFFER #1. If any I/O error is detected on Block #232 the status of the buffer changes to 'ERROR'. When in the normal course of events Block #232 is first accessed, an appropriate fetch value is returned (depending on whether the communicate was conditional or not). Further action depends on the user job or interpreter.

When a file is closed, all outstanding buffers are flushed.

STREAM OPERATIONS

In STREAM mode, if the length of the data requested is greater than the length available in the buffer, then only that length is transferred. The CPA is also modified (Length to transfer is decremented by, and work area offset incremented by, the length actually transferred) and a Restart fetch value (@FFFFFF@) returned to the interpreter. The communicate is re-issued as many times as necessary to achieve the originally specified transfer.

INDEXED I/O

General

All communicates requesting operations of files with INDEXED organization are routed to a separate section of LOGICAL I/O. OPEN and CLOSE communicates will always lead to the issuing of two separate communicates. A communicate is required to OPEN/CLOSE the key file and the other to OPEN/CLOSE the data file. All other communicates are separated into a SEARCH operation or operations, which are issued to PHYSICAL I/O, and a normal communicate either on the key or the data file which is rerouted into the main LOGICAL I/O routines. All INDEXED files have an extended FIB to deal with the key file. Note that there is no hardware search facility on any of the disk controls available with B 1800 systems. Consequently, a hardware search will require a physical read of all disk sectors, followed by a software comparison. On very large key files therefore, the performance of B 1800 CMS might be inferior to that of CMS hosts which have a hard disk search.

READ/WRITE COMMUNICATES

Sequential Access

READ	Determines the lowest key by merging Index and Overflow regions with priority assigned to Index, and generates a READ Random on the Data file.
WRITE	Only allowed when creating a new file. Generates a WRITE Sequential in the Index region and a WRITE Random in the Data file, after the last record previously written. (The keys must be in ascending order but duplicate keys are allowed if specified).

Random Access

READ	SEARCH in the Rough Table to determine the appropriate Index region. SEARCH in that region until a key value is found that is equal to or greater than the key required. If not found in the Index region, then SEARCH in the Overflow region, until a key value is found that is equal to or greater than the key requested. If not found again, an INVALID KEY fetch value is returned, otherwise perform a READ Random in the Data file.
WRITE	SEARCH in the Rough Table and the Index region for a key value equal to the key requested. If found, an INVALID KEY fetch value is returned if Duplicate keys are not allowed. If not found, then SEARCH the Overflow region for a key equal to the key requested. If found, an INVALID KEY fetch value is returned if Duplicate keys are not allowed. WRITE the appropriate entry in the Overflow region after the last key that is less than the key following the one requested and move all keys higher than the key requested up. WRITE Random in the Data file after the last record previously written.

PHYSICAL I/O

INTRODUCTION

PHYSICAL I/O is the lowest level of the MCP. It is the only part of the MCP that communicates directly with physical devices. Its basic function, therefore, is to translate requests for specific I/O action on files into direct operations on physical devices. It is responsible therefore for mapping user-specified files on to physical devices, and for driving the physical devices themselves, and for monitoring continually the status of physical devices. The last function involves performing Automatic Volume

Recognition (AVR) when physical devices are made ready. PHYSICAL I/O is also responsible for directory maintenance on disk devices.

STRUCTURE OF PHYSICAL I/O

The RESIDENT portion of PHYSICAL I/O handles all Magnetic Tape functions, Disk Read/Write routines, all routines handling the ODT, the Physical Bootstrap and all I/O queuing, initiation and completion routines. All other functions are in overlay segments with basically one segment performing one function (for example, DISK OPEN OLD, DISK OPEN NEW).

PHYSICAL I/O also uses certain data structures which are created by the system initializer depending on the exact configuration.

PHYSICAL I/O DATA STRUCTURES

Introduction

Physical I/O operations on the B 1800 are performed by engaging a channel (of which there are 15 available) and entering into a two-way dialogue with the hardware device control located in that channel. The hardware device control translates instructions received from the central processor into actions on the device(s) it controls and returns data and/or a Result Descriptor which indicates the success or failure of the action and, in the event of failure, the reason(s) for the failure.

PHYSICAL I/O uses various structures to control the I/O subsystem as follows:

CHANNEL CONTROL BLOCKS
CHANNEL CONTROL WORD
DEVICE CONTROL BLOCKS
FILE CONTROL BLOCKS

Channel Control Block/Channel Control Word

At system initialization time, one CHANNEL CONTROL BLOCK (CCB) and one CHANNEL CONTROL WORD (CCW) is initialized for each channel to which a CMS supported device control is connected. Any channel to which a non-CMS device control is connected is ignored, as are channels with no control connected.

The CCB contains:

- status flags which determine whether the channel is busy (I/O in process), active (I/O queued) or under test and, for a disk device, whether it is a cartridge or pack, and for a magnetic tape, whether it is Phase Encoded or NRZ.
- the address of the routine which analyzes the Result Descriptor.
- pointers to the DEVICE CONTROL BLOCKS which are attached to this channel.
- static information about the channel configuration (number of device, tape exchange).

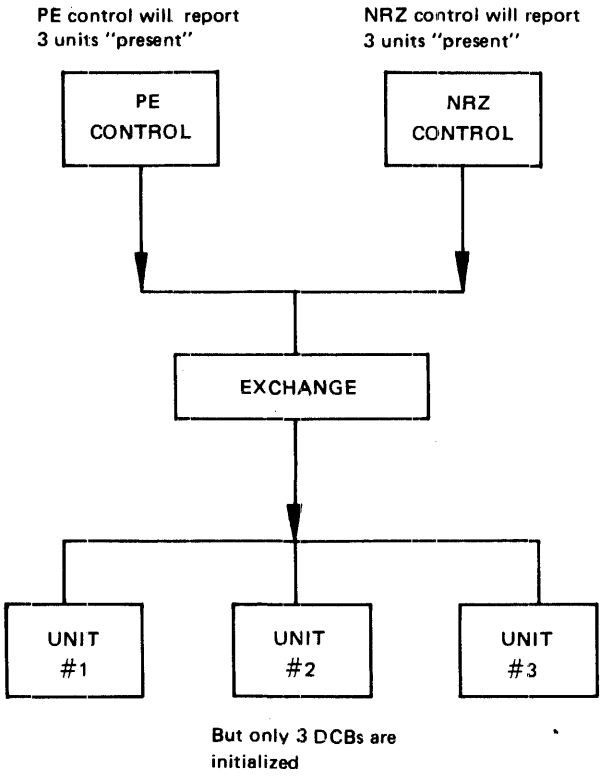
The CCW contains information which controls the physical transfer of data between the processor and the control on the channel. Basically this includes the transfer width, the memory buffer address and length and control flags.

Device Control Block

B 1800 CMS separates device controls into three categories:

1. Those controls that can only have one physical unit attached and are accessed serially. Devices in this category are card readers, line printers and the ODT.
2. Those controls that can have more than one physical unit attached but the units can only be accessed serially. This category includes all magnetic tape devices.
3. Those controls that can have more than one physical unit attached but a unit can be accessed serially and randomly. This category includes all disk devices.

At system initialization time, the DEVICE CONTROL BLOCKs (DCB) are created. For category (1) one DCB is created for each channel. For category (2) one DCB is created for each physical unit reported as 'present' by each control with the exception that, when physical units are connected via a tape exchange, only one DCB is initialized for each unit although each of the two controls reports that the unit is present.



For category (3) one DCB is created for each physical unit 'present'.

A DCB contains information about the physical properties of the unit as well as status information.

File Control Blocks

FILE CONTROL BLOCKs (FCB) are the main instrument used in the actual I/O operation. For serial devices and magnetic tape devices one FCB is created for each DCB, as the device can only be 'linked'

to one file at any time and therefore only one user. For disk devices each physical unit can accommodate many files and multiple users of any one file. Therefore a variable number of FCBs are created for disk, the number depending on the size of memory. The number of FCBs therefore limits the absolute number of I/O operations on disk that can be in process at any one time. The number of FCBs created is calculated according to the value contained in the LR register as follows:

LR = XXnnnn
NO.OF.FCB = XX + 32

So, if LR = 200000 (256K mem)
then the NO.OF.FCB = 32 + 32 = 64

PHYSICAL QUEUE MANAGEMENT

Introduction

A request for PHYSICAL I/O to perform an I/O operation involves four separate phases:

1. Analysis and validation of the request.
2. Queuing the I/O.
3. Initiation of the I/O.
4. I/O completion.

If the communicate is not valid, PHYSICAL I/O exits in Phase 1 without initiating the I/O. In Phase 2, PHYSICAL I/O queues I/O in the PHYSICAL activity QUEUE. Phase 3 analyzes the PHYSICAL ACTIVITY QUEUE and selects an I/O operation to initiate. In Phase 4 the results of the operation are analyzed. If an error occurred the I/O is queued again in the PHYSICAL ACTIVITY QUEUE for retry (each I/O is retried 16 times before an error is returned to the requestor). If the operation was successful the I/O is removed from the PHYSICAL ACTIVITY QUEUE and linked into another queue, the LOGICAL QUEUE which contains all I/Os that have been completed.

Physical Activity Queue

The basic element in the PHYSICAL ACTIVITY QUEUE is the FCB which represents one I/O operation. After validation of the communicate, the FCB for the file on which the operation is requested is 'primed' for the I/O operation required. (Note that in the case of an OPEN a free FCB must be found before the whole process can proceed). The FCB is then linked into the queue of I/O operations for the device on which the file resides. The DCB for that device contains the pointer to the first FCB queued. If no FCBs have been queued, the DCB itself is linked into a queue of devices with I/Os waiting, which is linked to the CCB for that channel.

The queue for any channel is examined to select the next I/O to be initiated as follows:

1. If no DCB is queued, no action is taken.
2. The DCB queue is examined. For each DCB in the queue, the FCB queue is examined. The first FCB on the queue that can be initiated is removed from the FCB queue and marked as 'active' (pointed to by the DCB).
3. The first DCB in the queue with an 'active' FCB is removed from the queue and marked as 'active' (pointed to by the CCB).
4. The 'active' DCB and 'active' FCB are used to 'prime' the CCW and the I/O operation is initiated.

Figure 10-1 illustrates the queue structure.

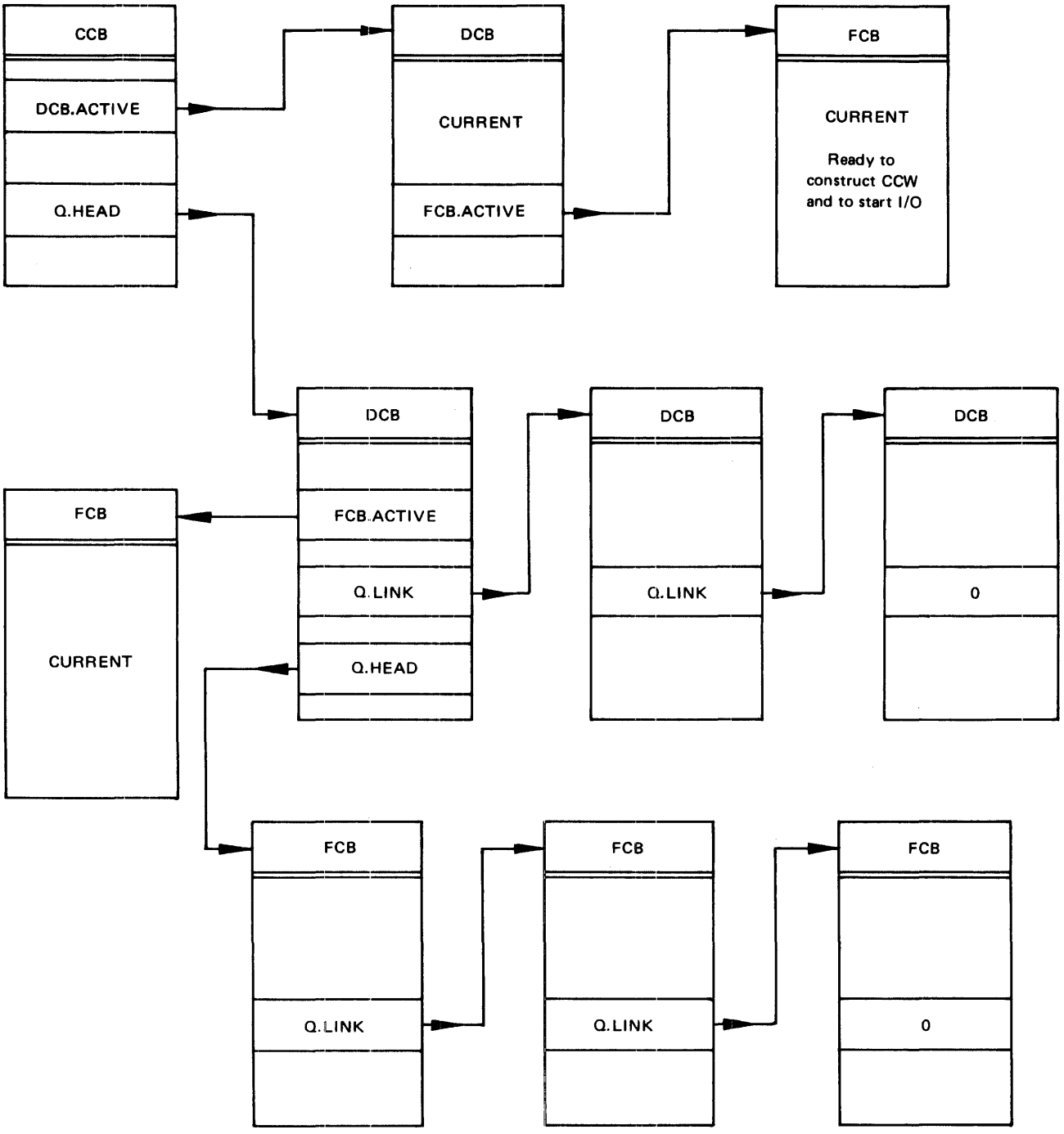


Figure 10-1. Physical Activity Queue

Logical Queue

When an I/O completes successfully or unsuccessfully (after retry) it is removed completely from the PHYSICAL ACTIVITY QUEUE and linked into the LOGICAL QUEUE which is a queue of all FCBs which have completed and for which, therefore, a result can be returned to the requestor. After linking into the LOGICAL QUEUE, the PHYSICAL ACTIVITY QUEUE routine is re-entered to fire off any I/Os that are waiting. If no I/Os are outstanding, the LOGICAL QUEUE is emptied.

When an I/O operation, after giving a CC(2) interrupt, has a Result Descriptor indicating a 'SEEK in Process', the device must not be de-allocated until the end of the operation for which the head positioning was required. But if there are some other devices on the same channel, they must be accessible during the time required for the SEEK. So, the channel is liberated but not the device: the same FCB remains active for the DCB.

When the SEEK is complete, the control issues another CC(2) interrupt, indicating that the head is positioned. At this stage the original I/O operation is queued during the I/O completion routine.

NOTE

DCC-1 control does not issue that CC(2) interrupt so a PAUSE instruction is sent every 4 milliseconds. This will give a CC(2) interrupt in order to allow PHYSICAL I/O to send a TESTOP instruction. This is done as long as the Result Descriptor indicating SEEK complete is not received.

PHYSICAL BOOTSTRAP

This routine in PHYSICAL I/O serves a dual purpose as it is used both to load the LOADER and to load overlays for PHYSICAL I/O.

Loading of LOADER:

Control is passed directly from the BOOTSTRAP routine in GLOBAL. The DFH (Disk File Header) of LOADER is accessed and the length of LOADER module in bytes is calculated. To this is added 800 bytes (for the EXMSG.TABLE and the maximum INIT.MSG) and a call is made on MEMORY MANAGEMENT requesting allocation of this amount of memory. MEMORY MANAGEMENT returns the absolute address of the start of the area of memory allocated. The file LOADER is now read from disk into this area. The entry for the MIX # in the Memory Assignment Table is changed to point to the area where LOADER has been loaded (the ICB of LOADER will be invoked instead of OS.ICB). The EX.MSG.TABLE and the INIT.MSG are copied into the area of memory immediately after the end of LOADER code. The MIX # (which is identical to the MAT index) is written into ICB.BASE.INDEX at level #0. The final action of this phase is to remove the suspension of MCP.SCL by resetting MX.BOOT.RUNNING in the MIX.TABLE.

Loading of a PHYSICAL I/O overlayed segment:

Each time a PHYSICAL I/O overlayed segment is needed, a working disk FCB is constructed and used by the OVERLAY.LOADER routine which loads the overlayed segment required in the Overlay Table (whose length is that of the biggest segment). After that loading the working FCB is released.

COMMUNICATES

The Communicate Verb in the FIB consists of two parts: the Verb and the Variant, which are each four bits long.

Verb	0	WAIT	Variant	0	NORMAL
	1	OPEN		8	WAIT
	2	CLOSE			
	3	READ			
	4	WRITE			
	5	SEARCH			

The Variant WAIT means that the required operation must be resolved before giving back control to LOGICAL I/O. Due to Asynchronous Communicates principles, only a READ and WRITE can efficiently use the variant (30 or 38, 40 or 48). The other Communicates are always constructed with the WAIT variant (08, 18, 28, 58).

Verb WAIT	Forces PHYSICAL I/O to wait for the completion of the first I/O operation requested for a file by a program before giving control back to the Interpreter. This is done by a flag meaning 'Program Suspended' in the appropriate FCB and a flag meaning 'Waiting I/O complete' in the appropriate entry of the MIX table.
Verb SEARCH	Reads the specified Key file of an
(only for Indexed Sequential)	Indexed Sequential file to check a given relation with a certain key. The data transfer of phase two is always bypassed.

MEMORY MANAGEMENT

INTRODUCTION

The strategy adopted for Memory Management is that of the variable-size partition. Each job when started is allocated a contiguous area of memory which contains its ICB, its LOCKED SLICE and the Virtual Memory Space within which code and data segments are loaded and unloaded. The aim of Memory Management is to keep one single area of available memory (at the high address end of memory) thus minimizing 'checker-boarding' the memory). Any change in the status of one partition, as for example, its demise or its enlargement, means that other partitions in memory may have to move.

The size of memory allocated to a partition varies during the lifetime of a task depending on two factors. The first is internal to the partition itself. If the amount of memory allocated initially proves, in the course of execution, insufficient to contain the 'working set' of the task, then that partition will be subject to 'thrashing'. When thrashing is detected, an attempt is made to remedy it by extending the size of the partition. The second is external to the partition itself and is concerned with the 'working set' of memory. If the absolute amount of memory available is insufficient to hold all the partitions of tasks currently executing, a measure of overall system thrashing occurs as partitions are rolled in/out or their sizes are reduced to accommodate new partitions for tasks commencing execution. A partition's size might then in the course of its life vary between the length of the ICB (reflecting a complete ROLL-OUT) and the actual size of all the tasks, data and code segments, ICB and Locked Slice.

Memory Management module consists of three parts which are scheduled separately:

VM	The management of virtual memory space within a partition. It is responsible for loading data/code segments. Handles ROLL-IN, COMPACTION and ROLL-OUT as directed by MM or WAKE.UP.
WAKE.UP	Decides whether or not to ROLL-IN a rolled-out program.
MM	Memory Management. Moves all the partitions to satisfy a request for memory or decides to COMPACT all partitions or to ROLL-OUT one of them when insufficient memory is available.

Three explicit Communicates are received from other modules:

Load a code segment (Verb #10)
Load one or more data segments (Verb #20)
Extend the size of a data segment (Verb #30)

When a program is started, a minimum size (length of LOADER + length of the LOCKED.SLICE of the program) is calculated by LOADER and a Communicate is sent to VM to ALLOCATE that size to the partition. At the end of the program's life, TERMINATOR sends a Communicate to MCP.VM to DEALLOCATE that partition.

The procedure followed when a request is received to load code or data segments is identical. Firstly the appropriate segment table is scanned and the size of all segments which are marked TO.B E.LOADED is calculated. If the total is less than the available space in the partition, then the required segments are loaded either from the Code file (for code segments and read-only data segments) or from the Virtual file (for read-write data segments) by issuing a Communicate to PHYSICAL I/O. In the case where the available memory in the partition is less than the total length needed, then MM is called to resolve the problem.

Extending a Data segment is only issued by LOGICAL I/O in the following circumstance: when LOGICAL I/O receives a Communicate to OPEN a file, it must construct in memory a Data segment for the FIB. This is done from information contained in the FPB. In the case where the buffer size is 0 (meaning that the value must be taken from the DFH), LOGICAL I/O assigns a size of 360 bytes for the file's total buffer space and then calls PHYSICAL I/O to perform the physical open. On return, the value of the buffer size has been updated by PHYSICAL I/O to reflect the actual value. If this value multiplied by the number of buffers is greater than 660, LOGICAL I/O has to call MCP.VM to extend the FIB by the difference.

VM handles the following functions when directed by MM:

COMPACTION

Only the segments IN.USE are retained in memory. For the other segments, all the READ/WRITE segments are copied to the Virtual file by issuing a Communicate to PHYSICAL I/O.
All segments remaining in memory are shifted upwards to provide only one available area whose size and location are available for MM.

ROLL-OUT

Only the ICB is kept in memory. All the other segments, including IN.USE segments, are released following the same principles as for COMPACTION and the LOCKED.SLICE is copied to the Virtual File.
All segments in use are marked 'TO BE LOADED'.
NOTE: When rolling-out the partition, VM computes the minimum size that will be necessary for ROLL-IN.

ROLL-IN

To process ROLL-IN, VM takes the following two actions:

1. Request from MM a partition extension (the memory space as computed when rolling-out).

NOTE: If MM cannot grant the space (no more programs to Compact or Roll-out), the Verb ROLL-IN is abandoned and the program concerned is put back at the tail of the Roll-in queue.

2. When the space has been granted by MM, the Locked Slice is first Rolled-in, then the data segments marked 'To be loaded' are Rolled-in, followed by the current code segment.

WAKE.UP

This entry is scheduled every second. It selects a program to roll-in (if there are programs rolled-out). A program is selected for rolling-in at an interval which is determined by the following formula:

11 − X seconds
where X = number of rolled-out programs up to a maximum of 6.

For example, if two programs are rolled-out, then each program will be rolled-in at 9 second intervals.

MEMORY MANAGEMENT

The rule of MM is determined by the effect of requests from VM on memory occupancy: it must decide what action to take in the event of a user partition requesting a resource which cannot be granted from its available memory. All available memory is kept as one contiguous area at the high address end of memory. A request to allocate memory from the available pool (as for instance at BOJ time) can be honored immediately with no perturbation of existing residents if, and only if, the amount of memory requested is less than the amount available. A request to de-allocate memory will cause no disturbance if, and only if, the area to be de-allocated is adjacent to the available area; otherwise partitions at higher addresses will need to be moved in memory in order to maintain just one area of available memory. A request to extend a partition likewise causes no disruption if the target position is adjacent to the available area of memory and that is sufficient to satisfy the extension request.

The actions available to MM, when insufficient memory is available for VM to honor a request to load segments, are:

- COMPACTION of all partitions
- ROLL-OUT of one partition: a program whose status is WAITING on some external event is deemed to volunteer for this action.

NOTE
ROLL-OUT has priority over all other actions.

MM examines all of memory and an instruction is transferred to VM which will first analyze it the next time it is scheduled for the appropriate MIX.

A few important points are outlined below:

MM is responsible for re-arranging memory occupancy (which it attempts to do asynchronously with other tasks). Any further requests which might disturb memory occupancy are refused until the present re-arrangement has finished.

A decision to roll-out one partition will only be taken when the COMPACTION of all partitions has finished and insufficient memory still remains.

A partition cannot be moved until all I/O operations outstanding for all files in that partition have completed. The reason for this is that PHYSICAL I/O has an absolute address for I/O buffers which are contained in the partition.

If the total amount of memory is insufficient to accommodate in total all partitions, then there will be system 'thrashing' and MM will take the decision to roll-out one partition. Thrashing is detected when three compactions have been done in 8 seconds or less.

When DP has been entered for a job, all the segments will be rolled-out to the Virtual file, in addition to the ICB and LOCKED.SLICE but the memory space is not released until the job has ended.

For a partition which has its size precisely defined in number of bytes by the execute command (memory size entered between braces), MM will honor the request if possible but the partition will not be subject to COMPACTION or ROLL-OUT nor will it ever be extended.

All interpreter partitions which have an invariant size are loaded at the low address end of memory; that is, when a job is started, if it needs the presence of an interpreter not yet in memory, the interpreter will be loaded after the last interpreter but before the first program in the Mix. This strategy can give more disturbances at BOJ time, but the interpreters will not be moved up or down in memory as long as there remains one user for them.

LOADER

This module handles the Beginning of Job for all System Utilities, Compilers and user programs started by the operator or zipped by another task.

This module has no resident segments permanently in memory and is loaded when needed in the partition which will be occupied by the job started. When loaded, LOADER is to all intents a normal task. The beginning of its code will be the copy of the ICB of the program to start. When its MIX is selected by the JOB scheduler, it will start to execute and attempt to load the the requested program and set its run-time environment.

PROGRAM FILE HANDLING

The first action of LOADER is to call PHYSICAL I/O to open the object file of the required task. If the designated file cannot for any reason be physically opened, the fetch-value returned is analyzed and the appropriate error message displayed. After that, a call is made on Memory Management to deallocate the memory space that was located.

If the file can be opened but is not a valid code file, then PHYSICAL I/O is called to close the program file and the action taken is the same as for an open failure.

LOCKED SLICE BUILD

The next stage is to call PHYSICAL I/O to read the PPB (Program Parameter Block) of the code file (this is always the first record of the code file). In the event of failure, the program file is closed and the action taken is the same as for open failure, except that no display is necessary since PHYSICAL I/O will already have displayed the appropriate error message.

The size of the LOCKED.SLICE is computed and a call is made to Memory Management to extend the partition by that amount. If a fatal result is returned, the program file is closed, an error message is displayed and LOADER exists as described previously. Otherwise, the LOCKED.SLICE is built from the code file in the area added to the partition.

The initiating message (if present) is copied immediately after the LOCKED.SLICE.

VIRTUAL MEMORY SPACE ALLOCATION

If an exact amount of memory is requested in the EX message, the space allocated will be that size with at least the memory space needed by LOADER + LOCKED.SLICE. In all other cases, allocation of memory is handled by Memory Management.

INTERPRETER LOADING

First a search is made in the INTERP.TABLE for the name of the interpreter contained in the PPB. There are three possible results:

1. Interpreter is already in memory.

In that case, LOADER goes straight to the next phase (VIRTUAL FILE CREATION).

2. Interpreter is being loaded into memory.

This means that once the interpreter has been loaded the action required is as described in 1. Therefore, the return address in Level 0 is set to a routine that checks 'Interpreter-Status' and LOADER waits, and the exits to the main idle loop in GLOBAL.

3. Interpreter is not in memory.

In this case, the interpreter's file-id is written into the INTERP.TABLE at the first available slot and the entry for the interpreter in the MAT has the flag INT.IS.LOADING set. A call is made on PHYSICAL I/O to open the file of the interpreter required. If no file exists, then an error message is displayed and LOADER terminates. For any other fatal failure the action is identical, except that no display is necessary.

Once located the file type of the interpreter is checked to ensure that it is a B 1800 microcode file. If it is not the correct type, the file is closed by a call on PHYSICAL I/O and LOADER terminates in the same manner as if no file has been found.

The memory size required for this interpreter is calculated and a call made on Memory Management to allocate a partition by that amount of memory and to update its entry point in the MAT.

The interpreter file is now loaded into memory by a call on PHYSICAL I/O. On successful completion the file is closed by a call on PHYSICAL I/O. The appropriate flags corresponding to the entry of the interpreter in the MAT are updated as well as the name of the interpreter in the Intepreter Table.

VIRTUAL FILE CREATION

The size of the VM file is now calculated as:

VMFILE.SIZE (in sectors)	(ICB.SIZE + LOCKED.SLICE.SIZE + CODE SEGMENTS + DATA SEGMENTS)/180 + Number of segments (because each segment must be on a sector boundary) + 20.

The VMfile is created in one area on the disk from which the program was loaded and the file-id is set to VMFILnn where nn is the MIX.

A call on PHYSICAL I/O is made to open the virtual file. In the event of failure, LOADER terminates as described before with or without a display, depending on the type of error.

TERMINATION

If the task priority class indicates that BOJ/EOJ messages are not suppressed, then the BOJ is prepared in the form 'BOJ PR = <class> TIME = HH:MM:SS' and displayed.

If this task was initiated by a ZIP with NO PAUSE, the appropriate fetch-value is returned to the initiating task so that it can resume.

Finally Level #0 of the ICB.CO-ROUTINE.LIST is changed so that the MAT.INDEX is now the interpreter so that the MAT.INDEX is now the interpreter and not LOADER, and the calling level and return address are set to 0. Now the LOCKED.SLICE and INIT.MSG (if present) are copied to their correct place in the partition, just after the ICB. The next time the MIX is scheduled control will be given to the interpreter which will start executing the task.

TERMINATOR

INTRODUCTION

This module handles the termination of a job.

A task can terminate in one of three ways:

1. Normal termination through a terminate communicate which was generated in the task by a compiler.
2. Input of a DS or DP command by the operator in response to a request by the interpreter as the result of the discovery of a fatal error.
3. Input of a DS or DP at any time during the task existence.

Normal Termination

An appropriate communicate is generated by the COBOL compiler whenever it encounters a 'STOP RUN' statement, or by the RPG compiler when an 'abort' condition is detected, and by the MPL/BIL compiler on encountering a 'STOP' statement. The interpreter calls LOGICAL I/O which recognizes the communicate as a TERMINATE and calls GLOBAL which changes the MAT.INDEX of level #0 to the index for TERMINATOR, sets the RETURN.ADDRESS for Level #0 to 0 and exits to the main idle loop in GLOBAL. The next time this MIX is scheduled, TERMINATOR will be executed.

DS or DP Required by Fatal Errors Detected by the Interpreter

When an interpreter detects a fatal error while interpreting S-code, it calls MCP.SCL to display the error message itself and a message indicating to the operator that the task should be DS'ed or DP'ed. It then calls GLOBAL which sets the flag WAITING.DS.DP in the MIX.TABLE entry for this task and exits to the main idle loop of GLOBAL. The effect of setting this flag is to disqualify the task from selection by the Scheduler and therefore completely inhibit its further execution. Only the entry of a DS or DP message is allowed and this forces the task to terminate.

DS or DP Requested by Input Command

MCP.SCL recognizes that DS or DP has been input and calls GLOBAL to reset the flag WAI TING.DS.DP and set DS.RUNNING (from this point an interpreter-directed DS/DP and an unsolicited DS/DP follow exactly the same path).

The MAT.INDEX for Level #0 is set to the MAT index for TERMINATOR and the appropriate entry points set for DS or DP and an exit made to the main idle loop of GLOBAL.

TERMINATION OF A JOB

There are three points in the TERMINATOR, which correspond to the function required:

1. Normal Termination
2. DS
3. DP

If the variant is DP or a normal terminate with the variant in the CPA set to create a dumpfile, then the CLOSEMODE of the Virtual file is changed from RELEASE to LOCK CRUNCH, the file-id is changed from VMFILnn to DMFILnn and a call made on MCP.VM to roll-out to the Virtual file the entire partition.

All terminate variants now follow the same path. The DST is scanned and all the data segments which are not FIBs have the flags TO.BE.LOADED, IN.USE and LOCKED reset. Similarly,for any code segments,the flags TO.BE.LOADED and IN.USE in the PST are reset.

All FIBs are now checked. If the FIB is not currently IN.USE then there are two possibilities:

1. The file is fully closed.

   In this case no further action is necessary.

2. The file is half-closed.

   PHYSICAL I/O is called to close the file fully. The IN.USE flag in the DST is reset and the internal number assigned to the file is released.

For FIBs that are marked IN.USE (the file is currently open) the same procedure is adopted for half-closed files except that LOGICAL I/O is called to handle the close, for flushing the buffers, and setting the end of file pointers.

When all user files have been closed, the Program file and the Virtual file are closed by a call on PHYSICAL I/O.

Calls are made on MCP.SCL to display any necessary messages: DMFILnn CREATED, JOB DS'ed, JOB DP'ed, EOJ.

Finally the appropriate Fetch Value is returned to the initiating task if this task was initiated by ZIP and the initiating task still exists in the MIX. The TERMINATOR module then terminates by calling MCP.VM to deallocate the partition.

ODT/SCL HANDLER

INTRODUCTION

The ODT/SCL HANDLER is responsible for performing the following functions:

1. Analysis and validation of all operator-initiated and ZIP-initiated input messages.
2. Construction and/or display of all output messages.
3. Logging all input/output messages to the system log files (SYS-LOG-01 through SYS-LOG-nn).

INPUT MESSAGE HANDLING

Input messages can be received from two sources:

1. Operator-initiated
2. Initiated by a ZIP communicate

Both types of input follow a common path in that the contents of the message must be analyzed and validated for conformity to System Communicate language (SCL) syntax. All input messages from ODT are received in the first instance into a SPO input buffer. The ODT/SCL HANDLER resides in MIX #0 and, when scheduled, checks whether there is any input from the file or any output to the ODT. Messages from a ZIP are not physically passed to the ODT/SCL HANDLER, only pointers to the location of the message. Analysis of the message is concerned first of all to establish whether the request is for an intrinsic function (AX, ST, GO, OL, MX) or a request to load a job. In the first case, the message is analyzed for correctness and the appropriate output generated. If the request is a load request control, it is passed to the BOOTSTRAP routine in GLOBAL to load the LOADER which will analyze the load request.

OUTPUT MESSAGE HANDLING

The ODT/SCL HANDLER is responsible for four kinds of output messages:

1. Echoing of SPO input and ZIP DISPLAY'ed messages.
2. Output messages for intrinsics.
3. User job displays.
4. Error messages for errors discovered by other MCP functions.

For type (4), a coded message is passed to the ODT/SCL HANDLER from which the full message is constructed using the file MCP.SCL.ERRS.

The ODT.SCL HANDLER tanks output messages in a queue 20 deep. The queue itself is cyclic. Into this queue are copied all messages to be output and/or logged. If the queue is full (that is, all 20 messages are in the status waiting to be logged or displayed), the ODT/SCL HANDLER will not accept messages until messages in the queue have been cleared. In the event of a large stream of output messages which the ODT/SCL HANDLER cannot handle in the normal manner, the messages are logged and displayed from the log when a suitable pause in the stream is reached.

The ODT handling is as follows:

- The screen itself has 24 lines of 80 characters each.
- The first four lines are reserved for input. Whenever a message is displayed the first four lines are always cleared.
- The output messages are displayed on the remaining 20 lines using a First-In-First-Out (FIFO) principle. This means that the most recent message is displayed on the last line.
- Output messages are sent line by line to the ODT. Control characters are sent to 'scroll' the screen (that is, move all previously sent output messages up one line) for each line sent. Therefore, the screen will always display the 20 most recent messages (if all messages are one line only).
- If historic messages are required, a 'recall' facility with paging forward and backward is available (though this is handled by SYS-SUPERUTL).

LOG MANAGEMENT

The ODT/SCL HANDLER is responsible for the physical logging of messages to disk and management of the log files themselves. For the B 1800 CMS implementation, three log files are built at Clear/

Start. Whenever a log wraparound is likely to happen, the ODT/SCL HANDLER will send a warning message. However, it is the operator's responsibility to save the log files, if desired, by a TL command.

SECTION 2
FILES AND MEDIA

INTRODUCTION

CMS is a disk based system, but supports many other peripherals. All peripherals are accessed by the MCP via files. In this section, the physical and logical characteristics of each medium are discussed. To avoid repetition, common definitions are listed below.

- A physical file refers to a file as it resides on the medium.
- A logical file refers to a file as it is seen from a program.
- Under CMS, all data is internally represented according to the ASCII (American Standard Code for Information Interchange) collating sequence.
- A physical record is the smallest unit which can be accessed on the medium (see figure 2-1).
- A block is the unit which is transferred to a program buffer by an I/O routine; that is, it is the unit of a physical I/O operation. It consists in general of one or more physical records. It must start at the beginning of a physical record. Where it ends is medium dependent, but any further bytes in the last physical record of the block are ignored (see figure 2-1).
- A logical record is the section of a file which is made visible to the program as a result of an I/O communicate; that is, the unit of a logical I/O operation. There may be one or more logical records in a block but each logical record must be completely contained within one block. A record is made visible to the S-program in a work area data segment (see figure 2-1).
- A label is a block of identifying data which may exist on any CMS file medium. It is used for identification, and so that any physically compatible device can handle a particular file.
- Bits are numbered from 0 upwards, left to right.
- The MCP maintains a configuration table (CT) which shows the type of each device (table 3-3), what it is assigned to, and either an id (name) extracted from the label on the medium, or indications that the medium is scratch or unlabelled, or unavailable. When a medium is placed in a device capable of input, the MCP attempts to read the label at the start of the medium, extracting its id for the CT. When a medium is placed in a device which is not capable of input, the CT entry is marked scratch. When any medium is dismounted, the CT is marked not ready. These automatic actions are referred to as Automatic Volume Recognition (AVR).

THE DISK MEDIA

A significant aspect of CMS design is the fact that it is disk oriented. The physical structure of a disk and its logical layout under CMS are described separately.

DISK PHYSICAL CHARACTERISTICS

The discussion that follows applies to all disks supported by the CMS MCP, except the Industry Compatible Mini Disk (ICMD) disk which is only accessed sequentially.

Disks are capable of storing data on both sides. They consist of many recording cylinders, each with one track per surface. Each track is further divided into sectors (or segments) and each sector is capable of storing 180 bytes of data.

Figure 2-2 illustrates the read/write mechanism on these disks.

The number of cylinders per disk depends on the individual disk type. Table 2-1 shows the various types supported by the MCP and some of their attributes.

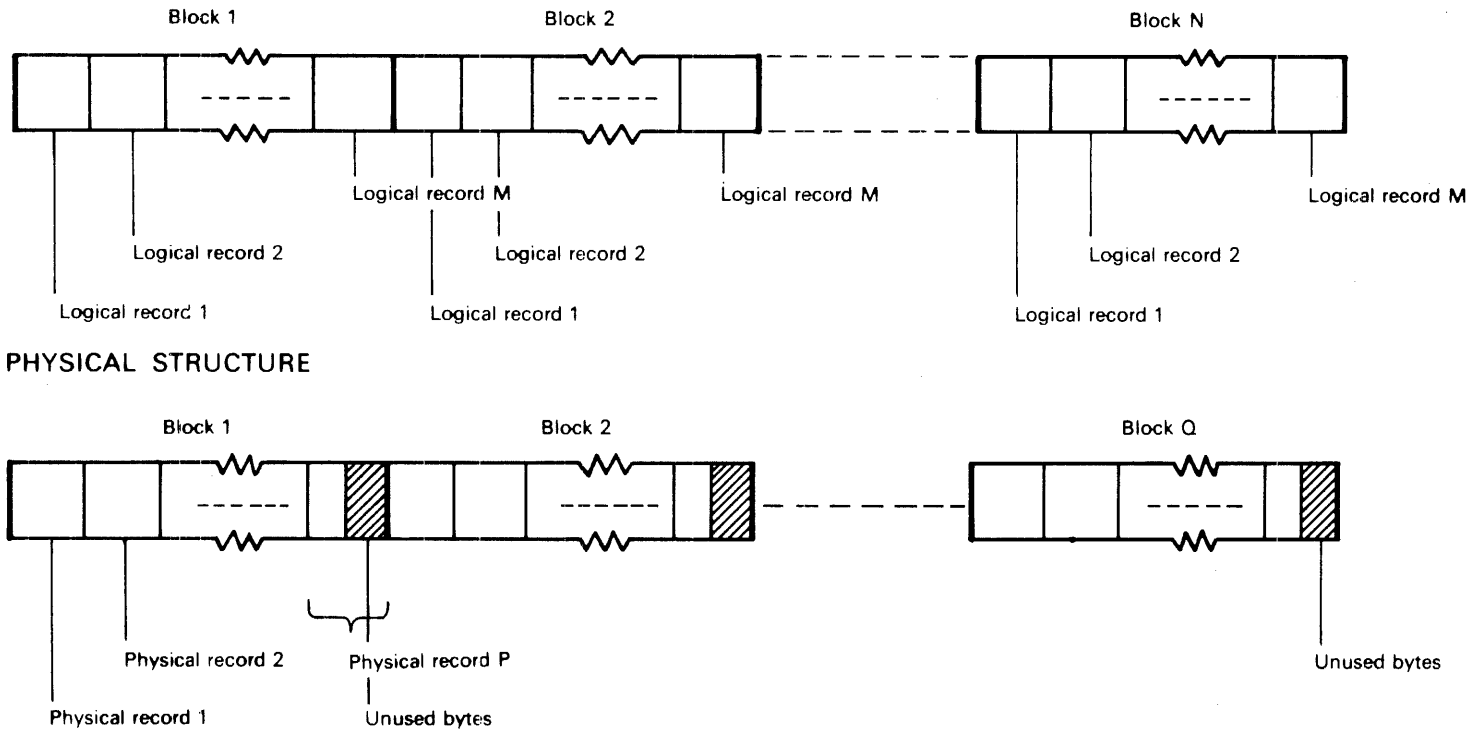


Figure 2-1. Logical and Physical Structures

More details can be found in [5].

A physical record on a disk is one sector. A block can contain any number of bytes and can therefore end in the first, or any subsequent, physical record.

DISK LOGICAL CHARACTERISTICS

Before a disk can be used under CMS, it has to be initialized using either the DSKUTL utility or the Disk Initialize 'IN' stand-alone utility [1]. During this process, the user assigns a serial number, a disk identifier (disk-id), owner's name, date, and the maximum number of files that can be stored on the disk (must be less than 2805), which determines the directory size. The basic function of initialization of a disk is to assign an address to each sector so that it can be addressed by the MCP. The 'IN' also generates a table for future identification of the disk, an empty disk directory, a table of available areas on the disk for future allocation, and a bootstrap to provide the micro-routine necessary to load the system into memory and activate it (warm start). From there, the MCP has the responsibility of maintaining the directory, allocating disk space, addressing records, and managing virtual memory. Track 0 contains information controlling subsequent access to the disk and must be error free. The layout of this track is illustrated in table 2-2.

Table 2-1. Disk Types and Attributes

Disk type	Capacity	No. of surfaces	No. of cylinders	No. of tracks	Sectors/ track	Bytes/ sector	Average access
Cartridge	2.3 MB	2	203	406	32	180	145 ms
Cartridge	2.3 MB	2	203	406	32	180	80 ms
Cartridge	4.6 MB	2	406	812	32	180	100 ms
BSMI	1.0 MB	2	88	176	32	180	325 ms
BSMII	3.0 MB	2	136	272	59	180	157 ms
ICMD	243 KB	1	–	75	26	128	343 ms
Fixed-201I	9.2 MB	2	406	812	64	180	55 ms
Fixed-201I	18 MB	4	812	1624	64	180	55 ms
Fixed-201I	27 MB	6	1218	2436	64	180	55 ms
Fixed-201I	37 MB	8	1624	3248	64	180	55 ms
Fixed-211	20 MB	2	335	1340	80	180	53 ms
Fixed-211	40 MB	4	335	2680	80	180	53 ms
Fixed-211	80 MB	8	335	5360	80	180	53 ms

Table 2-2. Track 0 Layout

Sector	Contents	Related Notes
0	Label	see figure 2-3 and table 2-4
1	Security information	see note 1 below
2-26	B 90 Bootstrap/system dependent	see note 2 below
27	SQ utility recovery work area	
28-29	Reserved	
30-31	Bad area log	see below

Related notes :

1. This sector is to contain information which could control access to the disk.
2. The B 90 Read Only Memory (ROM) contains a disk bootstrap routine which searches for a disk which contains this bootstrap. This search begins with the disk whose controller has the highest channel number. When a disk is found that contains a bootstrap, the boostrap is loaded and entered. When PK3 is pressed the bootstrap initiates another search for an 'MCP' file. If an MCP file is found, it is loaded into memory, and that disk is treated as a system disk. If no MCP is found on that disk, the search for a bootstrap on another disk is initiated. The resulting content of memory, and consequent actions (Warm Start), are described in Section 7.

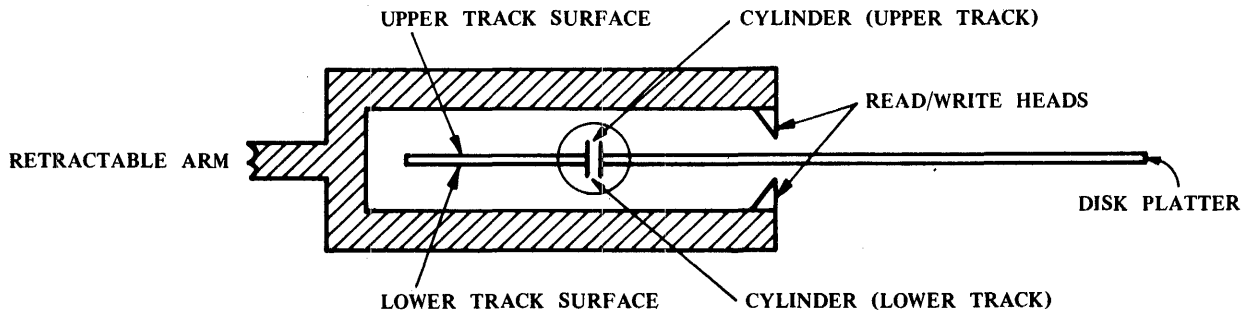


Figure 2-2. Disk Read/Write Mechanism

Table 2-3. Bad Area Log

Bytes	Contents
0-1	Total of bad allocation units recorded in this log
2-31	Binary zero. Reserved for possible future expansion
32-33	Address of first of group of contiguous bad allocation units
34-35	Number of allocation units in group
36-359	81 more address/length pairs

Disk Labels

The CMS programmer references disk files by their symbolic names. The MCP resolves these names to actual physical disk addresses. The file identifier has two components: a disk name (disk-id) and a file name (file-id). The disk-id is a 7 character field, and the file-id is a 12 character field (see also the FPB, table 3-1). A disk-id of '0000000' (seven zeros) is reserved for identifying the current system disk, and is used by default if the disk-id is not explicity stated.

Disk files are either permanent or temporary. A permanent file is one whose name appears in the disk directory and which may be accessed at any time by any program. A temporary file is one created by a currently running task and its name does not appear explicity in the disk directory. A temporary file has a directory entry with a file-id indicating a temporary file and the task using it has a pointer to enable its own access to the file. When a new file is opened, a link is forged between that file and the task. This link provides the only pointer to a new file. When a new file is closed, three options are available. The file can be purged, the disk space de-allocated and the link between task and file dissolved. The file can be entered into the disk directory making it an old file and dissolving the link between the task and file, or finally, the file can be left as a new file with the link intact, in which case the file is regarded as 'half-closed'.

Disks are labelled compatibly with Burroughs Interchange format which requires certain fields to be represented using the EBCDIC (Extended Binary Coded Decimal Inter-Change) code. The various fields in the label are illustrated in table 2-4. A sample label is exhibited in figure 2-3.

```
E5D6D3F1F0F0F0F0F2F740E2D3F9C9D5E3C5D9D5D352454C492E3120E2F9F000
0000000004954494F2020202020202020200000000000000000000000000000
0000000000000000000000000000040E5D6D3F2F2F9F2F7F6404040C2C4E220
00CB02200A00005F1F000040006E0100006900000000000F0F0F0F0F0F0F0F0
F0F0F0F0FF0000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000
```

Figure 2-3. A Sample Disk Label

Table 2-4. Disk Cartridge Label

Field Content	Location (decimal)	Length (bytes)	Recording mode
'VOL 1'	0-3	4	EBCDIC
Serial number	4-9	6	EBCDIC
Blank	10	1	EBCDIC
'SL9INTERNL'	11-20	10	EBCDIC
Cartridge identifier	21-27	7	ASCII
'S9' (system interchange code)	28-29	2	EBCDIC
Zero	30	1	EBCDIC
Reserved	31-36	6	
Owners identification	37-50	14	ASCII
Reserved	51-78	28	
Blank	79	1	EBCDIC
'VOL 2'	80-83	4	EBCDIC
Initialization Date (YYDDD)	84-88	5	EBCDIC
Initializing System (for example 'B90')	89-94	6	EBCDIC
Reserved	95	1	ASCII
Number of Cylinders	96-97	2	BINARY
Number of tracks per cylinder	98	1	BINARY
Number of sectors per track	99	1	BINARY
Number of sectors for file directory name list	100	1	BINARY
Sector address of directory name disk	101-103	3	BINARY
Number of sectors for available table	104	1	BINARY
Sector address of available table	105-107	3	BINARY
Maximum number of files	108-109	2	BINARY
Unit of allocation (in sectors)	110	1	BINARY
Sector address of first file header	111-113	3	BINARY
Reserved	114-118	5	
Integrity Flag (0 = OK)	119	1	EBCDIC
Actual error count	120-125	6	EBCDIC
Bad sector count	126-131	6	EBCDIC
Reserved for MTR	132-135	4	
Sector address of PPIT	136-138	3	BINARY
Length of PPIT	139	1	BINARY
Logical unit number	140	1	BINARY
Reserved	141-179	31	

Note that the label contains an integrity flag which is set when the disk is loaded and reset when the disk is unloaded by the MCP. Therefore, the flag remains set if a system failure occurs. If a disk is loaded with its integrity flag set, the MCP automaticially de-allocates all temporary areas by rebuilding the Available Table as the complement of the File Directory Name List.
The File Directory Name List, the Available Table and File Directory Header List are described in the following paragraphs.

The Disk Layout

As illustrated in table 2-4 and in figure 2-3, the disk label contains pointers to:

1. File Directory Name List. This is a table which contains an entry for every file on that disk (table 2-5). Each entry contains the name of that file, and the sector address of its Disk File Header (DFH).
2. An Available Table. This is a table which contains pointers to all unallocated and available areas on the disk (table 2-6).
3. The first Disk File Header (DFH). This is a direct link from the label to the File Directory Header List which is a list of DFHs; one for each file on the disk. Each DFH, in turn, contains pointers to those areas that have been allocated to that file (table 2-7).
4. The Pseudo Pack Identifier Table (PPIT). (See table 2-9). Pseudo Packs are a means of accessing a number of physical disk units as one or more logical disk units in a structure known as a Fixed Disk Assemblage (FDA). An FDA is a collection of pseudo packs residing on one or more physical packs.

For example, a system may have an 18 MB non-removable 2011 drive which consists of two physical disk drives; it may be desirable to have a larger number of named disks on the system. This is achieved by creating a number of pseudo packs which can reside on either or both physical drives, but are seen by the system as a number of differently named disk units. In this way, the example system with only two disk drives may appear to have a much larger number of disk drives by using pseudo packs. The B 90 system does not implement pseudo packs.

The entire Available Table, the File Directory Header List, and the File Directory Name List must be contained, with nothing else, within an integral number of tracks. The complete structure is allocated as a contiguous group of tracks, preferably in the same cylinder. The Available Table is placed in the first sectors of these tracks, followed by the File Directory Name List. The remainder of these tracks that have been used, and all other tracks allocated to the directory, contain the File Directory Header List. It is a requirement that the complete directory must be confined to the part of a disk with sector address 2^{16}. The maximum number of files on a disk, and therefore the File Directory Header List size, is 2805.

Disk File Directory Name List

This is a contiguous block of disk sectors whose number depends on the maximum number of files that can reside on that disk, as declared at initialization time. The number of sectors required is calculated on the basis of one 16-byte entry per file, and 11 entries to the sector. When the number of sectors is decided, it cannot be altered without re-initializing the disk with attendant loss of data.

The disk File Directory Name List is pointed to, and its length (in sectors) is described in the disk label. The format for each sector, and a sample File Directory Name List associated with figure 2-3, is illustrated in table 2-5 and figure 2-6 respectively.

Table 2-5. File Directory Name List Sector

Content	Location (Decimal)	Length (Bytes)	Recording Mode
File Identifier	0-11	12	ASCII
Pseudo pack tag	12	1	ASCII
Directory Index	13	1	BINARY
DFH Sector Address	14-15	2	BINARY
10 further 16 byte entries	16-175	160	ASCII/BINARY
Zeros	176-179	4	BINARY

When a new file is opened, a slot in the directory is reserved for it, the chosen slot being the lowest address available slot in the name list. The name of the new file is not entered into the chosen slot in the name list unless the file is closed with LOCK. During the period between the opening of a new file and its insertion into the permanent directory, 81 is placed in each byte of the slot's name field (the first 12 bytes). Entries corresponding to future files contain 80 in the first 12 bytes and bytes 13-15 contain the sector address of the directory entry as seen in the example in figure 2-4.

The directory index gives the position of this entry within the directory, starting at 0.

Each sector of the File Directory Name List can hold 11 entries. It is likely that the maximum number of files requested at initialization time is not a multiple of 11, and therefore the final sector of the name list may contain some unusable entries. These are indicated by 82 in each byte of the name field.

```
5359534D454D2020202020202000006942494C494E5445525020202020001006A
434F424F4C494E5420202020202002006B4D435020202020202020202020003006C
534F525420202020202020202004006D534F5254494E5452494E53202005006E
414D454E4420202020202020202006006F434820202020202020202020020070070
434845434B4144554D5020202008007143 4F5059202020202020202020090072
435245415445202020202020200A007300000000
465320202020202020202020200B00744B412020202020202020202020200C0075
4C442020202020202020202020200D00764C495354202020202020202020200E0077
4C522020202020202020202020200F00784D4F444946592020202020202020100079
504D423830202020202020202011007A435020202020202020202020202012007B
584420202020202020202020202013007C504420202020202020202020202014007D
504720202020202020202020202015007E00000000
524D202020202020202020202016007F544150454C522020202020202020170080
544150455044202020202020201800815550444154452020202020202020190082
44554D50414E414C5953452020 1A0083434F2020202020202020202020201B0084
434F425356455254455220202 01C0085434F424F4C312020202020202020201D0086
434F424F4C322020202020202020201E0087434F424F4C332020202020202020201F0088
434F424F4C342020202020202020202000890000000 0
434F424F4C352020202020202020202100 8A434F424F4C362020202020202020220088
434F424F4C372020202020202020230 08C434F424555255320202020202020240080D
434F424F4C472020202020202020250 08E525047504841534531202020202600 8F
525047504841534532202020202027009052504750484153453320202020280091
525047504841534534202020202900925250474E414D455320202020202A0093
525047535 94E544158202020202B009400000000
525047534B454C2020202020202C009552504756455253494F4E2020202D0096
525047455 52F5253202020202E0097504D2E4D2E492E312E312020202F0098
504D2E4F2E492E312E312020203000994D504C372E312020202020202031009A
4D504C372E322020202020202032009B4D504C372E332020202020202033009C
4D504C372E342020202020202034009D4D504C372E455252202020202035009E
4D504C372E534547202020202036009F00000000
4D504C372E53594E2020202020370 0A04D504C372E4D41535420202020380 0A1
4D504C372E4C495354532020203900A24D504C372E4C494220202020203A00A3
434F4E56455253494F4E2020203B00A4535452494E472020202020202003C00A5
444543494D414C2020202020203D00A64D414B45202020202020202020203E00A7
4A45414E494520202020202020 3F00A8808080808080808080808080808020400064
808080808080808080808080808020410064000000000
808080808080808080808080808020420065808080808080808080808080808020430065
808080808080808080808080808020440065808080808080808080808080808020450065
808080808080808080808080808020460065808080808080808080808080808020470065
808080808080808080808080808020480065808080808080808080808080808020490065
808080808080808080808080808020 4A0065808080808080808080808080808020 4B0065
808080808080808080808080808020 4C006500000000
```

Figure 2-4. Sample Disk Directory Name List corresponding to the label in Figure 2-3.

Disk Available Table

This is a contiguous block of disk sectors whose number depends on the maximum number of files to reside on that disk, as declared at initialization time. Each entry in this table corresponds to an unallocted area on the disk. Entries are added to this table, or present ones expanded, when a file is purged and its areas de-allocated. Entries are deleted from it when a file is opened. Each entry is 6 bytes long, which means that 30 entries reside per sector, the last being a string of binary 0's.

The number of sectors required to contain this table is chosen to be

$$3 + \left[\frac{\text{maximum number of files}}{16}\right]$$

where [+] denotes the integer portion of the function.

Entries are not sorted in any manner. The disk space is combined with any adjacent free space to minimize the number of Available Table entries.

Bytes 4 and 5 of the first entry, and the corresponding bytes in subsequent entries contain the address of the last allocation unit in the available area + 1, so that when an area is de-allocated (that is, put in the available table), the MCP scans the current available table for an end address which is equal to the start address of the area to be de-allocated. If found, the MCP amalgamates the new area with the old available area.

Table 2-6 illustrates the content of a sector of the Available Table while figure 2-5 exhibits, as an example, the first sector of the Available Table that corresponds to the example of figures 2-3 and 2-4. The remaining sectors contain unused entries.

Table 2-6. Available Table Sector Format

Content	Location (Decimal)	Length (Bytes)	Recording Mode
Length of available area in allocation units	0-1	2	BINARY
Address of the first allocation unit in the available area	2-3	2	BINARY
Address of the last allocation unit in the available area + 1	4-5	2	BINARY
28 Further 6 byte entries	6-173	168	BINARY
Zeros	174-179	6	BINARY

```
00C4152B15EF0000000032C01CCF15F132C0000000010000005114D9152A0000
0001000000000001000000000001000000000001000100000001000100000001
0001000000010001000000010001000000010001000000010001000000010001
0000000100010000000100010000000100010000000100010000000100010000
0001000100000001000100000001000100000001000100000001000100000001
0001000000010001000000010001000000000000
```

Figure 2-5. First Sector of the Available Table corresponding to Figures 2-3 and 2-4.

In addition to the available areas, entries corresponding to unused entries, bad areas (those containing only bad sectors), or ghost areas (that part of the virtual addressing range which extends from 0 to FFFF which is not physically available on a particular disk) are differentiated. This designation is accomplished as follows:

Unused entries: Length= 0, start address = 1.

Bad Areas: Length = 0, start and end addresses exchanged.

Ghost areas: Same as bad sectors.

File Directory Header List

The File Directory Header List is a list of Disk File Headers (DFHs), one for every file on the disk, and forms a contiguous block of sectors whose start address and size are indicated by fields in the Disk Cartridge Label. Each sector holds one DFH, and the format of a DFH is illustrated in table 2-7. A sample DFH for the first file corresponding to figures 2-3, 2-4 and 2-5 is shown in figure 2-6. The total number of sectors allocated to the header list equals the maximum number of files declared for that disk at initialization time, and therefore also the number of entries which must be allocated to the File Directory Name List. The ordinal position of a file in the header list is the same as its position in the name list.

When a new file is created, an available entry in the directory is allocated and the file name is set to indicate a temporary file. As disk space is allocated to the file, the file header is updated on disk to reflect the allocated space. When the file is closed with lock, the directory entry is updated to reflect the new file name and file size.

```
53595534D454D2020202020202020200000000032393237363736333239000B40001
0001000032C039393901 32C0000000000000000000000002080000000000032C0
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000
```

Figure 2-6. DFH of SYSMEM corresponding to Figures 2-3, 2-4 and 2-5.

Table 2-7. Disk File Header Format

Content	Location (Decimal)	Length (Bytes)	Recording Mode	Related Notes
File Name (same as in File Directory Name List)	0-11	12	ASCII	1
Spare (Blank)	12	1	ASCII	
File Type (see table 3-2)	13	1	BINARY	
Flags	14-17	4	BINARY	2
Generation Date (YYDDD)	18-22	5	ASCII	3
Last Access Date (YYDDD)	23-27	5	ASCII	3
Record Size (in bytes)	28-29	2	BINARY	4
Number of records per block	30-31	2	BINARY	5
Number of sectors per block	32-33	2	BINARY	
Implementation Level number	34	1	BINARY	6
Maximum file size in records	35-37	3	BINARY	
Save Factor (0-999)	38-40	3	ASCII	
Maximum area in use (0 = None)	41	1	BINARY	
Number of records in last area	42-43	2	BINARY	
Generation number	44-45	2	BINARY	7
Number of spare bytes in last record (stream I/O)	46-47	2	BINARY	
Cartridge name of overflow pack	48-54	7	ASCII	8
User count (see table 2.8)	55	1	BINARY	
Area bit map 1	56-57	2	BINARY	9
Area bit map 2	58-59	2	BINARY	9
Address of file area 1 (in allocation units)	60-61	2	BINARY	10
Size of file's area 1 (in allocation units)	62-63	2	BINARY	10
14 further 4-byte entries for 14 more areas	64-119	56	BINARY	
Address of file area 16 (in allocation units)	120-121	2	BINARY	
Size of file's area 16 (in allocation units)	122-123	2	BINARY	
Host machine dependent link to overflow pack	124-125	2		8

Related notes:

1. This field will contain 80 for an available unused header sector, and 81 for temporary (currently new) files. This is the same in corresponding entries in the File Directory Name List. The remaining bytes remain undefined.

2. Flags are as follows:

   Bit 0 set = file has been 'crunched'
   Bit 1 set = rough table valid
   Bit 2 set = file resides on two volumes
   Bit 3 set = single area file
   Bit 4-31 = reserved

3. This field is moved to the corresponding field within the File Parameter Block (FPB) (Section 3) when the file is opened and can therefore be accessed programmatically. It is moved back to its location in the DFH at close time, providing that the appropriate bit in the FPB flags field is set.

4. The record size must be specified when creating a new file (in the FPB) and is then copied into this location in the DFH. Subsequently, this value is used whenever that file is opened with the corresponding field in the FPB being zero.
5. The block size must be specified when creating a new file. It is then copied into the location in the DFH. Subsequently, this value is used whenever that file is opened with the corresponding field in the FPB being zero.

6. This value is used by the system to ensure that only files compatible with any paricular release are handled by that release. This is done by comparing this value with the corresponding value in the file's FPB.

7. Whenever any file is opened, the generation number (GN) field from its DFH is moved to its FPB GN field, which facilitates the programmatic inspection of GN's values. If that file is opened with output capability, then that number is incremented by 1 at close time. If the appropriate bit in the flags field within the FPB is set, the GN in the FPB is moved to the GN in the DFH when the file is closed. This enables the program to set the GN field explicitly. This facility is most relevant when dealing with Indexed files. As described in Section 3, an Indexed file consists of two related files: a data file and a key file. At open time, the GN fields in the two DFHs are compared (if the 'check' bit in the flags field within the Indexed file's FPB is set), and if not equal, the open fails. This mechanism protects against opening an incompatible file pair, as could be the case if the data file is modified to alter the keys in some records. When the SORT utility is requested to create a new key file from an already existing file, the SORT picks up the GN from the FPB of the data file (which is a copy of the corresponding value in the DFH) and moves it to the GN in the FPB of the key file and therefore, once closed, to the corresponding field in the DFH of the key file. This ensures initial compatibility between the key file and the data file.

8. Dual disk files are permitted and are then symmetrical between the two disks. The file name appears in both File Directory Name Lists and there is a DFH on each disk. Each DFH contains the name of the other disk and contains two bit maps of the file areas (see note 9). Because of the symmetry, neither disk can be regarded as master. However, both disks must be on line whenever the file is in use.

9. Area bit maps are 16 bit fields in which each bit represents one of the 16 possible file areas. The most significant bit in the bit map corresponds to the first area, and the least significant bit to the 16th area. The interpretation of each map bit setting is as follows:

   Area bit 1: 1 means that this area is allocated on this pack. 0 means that this area is not allocated on this pack or is on an overflow pack.

   Area bit map 2: 1 means that this area is allocated on the other pack. 0 means that this area is not allocated on the other pack or is on this pack.

10. Addresses and sizes of file areas are in terms of the allocation units of this cartridge. This is fixed at initialization time as an integer multiple of sectors. Addresses for areas on overflow pack are not necessarily correct. Sizes for areas on an overflow pack are correct and are given in terms of the allocation unit of this pack.

Note that every CMS disk directory contains a file describing the entire physical disk. The file has a name of SYSMEM, and a special file type which prevents access except to tasks of special priority setting as indicated by the priority field in their program parameter block. The DFH of SYSMEM corresponding to the examples in figures 2-3, 2-4 and 2-5 is shown in figure 2-6.

Table 2-8. User Count Field Format

Bits	Content
0-2	Total number of users (7 = locked)
3	File opened shared
4	File opened lock access
5-7	Number of users

Note: If bit 3 is set, then there is a shared user count. If bit 4 is set, then there is a lock access user count. Bits 3 and 4 are mutually exclusive.

Shared Disk Files

CMS permits a single disk file to be accessed by several users simultaneously. The maximum number of concurrent users is seven. However, a user can specify (through the open adverb field in the FPB) whether or not he is willing to share access to the file with other users. Each separate OPEN of the file creates a separate File Information Block (FIB) within the OPENing task. This data segment is constructed by the MCP at file open time to contain pointers to the File Control Block. At OPEN time, the value of the End of File (EOF) pointer and the disk addresses of all currently allocated file areas are copied into the FIB/FCB. If a non-shared user extends the file either by writing records beyond the existing EOF pointer, or by randomly writing records into disk areas that were previously unallocated, then these extensions are only recorded in the users FIB. The values in the DFH are not updated until the user closes the file. Therefore, any records that are added in this manner are not visible to users who open the file before the non-shared writer closes it. Note however that new records, added within the range of the original EOF pointer, which lie in a previously allocated file area do become visible when the writer's buffer has been copied to disk and any readers have then performed an appropriate actual disk read. Any request to open the file non-shared with output capability, will acquire the data file with lock access, which means that only input users can access that file. A non-shared key file is opened with LOCK meaning it is not available to other users.

A request during close to crunch a file is ignored if it is currently shared.

Disk Space Allocation

Space for a disk file is allocated in up to 16 separate areas. An area is not allocated until there is an indication that it is required. Therefore, an area is allocated on the first logical write attempt to that file whether or not this causes a physical access.

The address and size of each disk file area are recorded in the DFH. The values of the number of sectors that are allocated, in due course, are calculated at creation time. The sizes of the individual areas are not necessarily equal.

The algorithm for deciding the number of disk sectors to allocate to each area is decided according to the following rules:

1. For a normal data file the size in sectors is the larger of 256 x allocation unit, or

$$\frac{1}{16}\ \frac{\text{(maximum file size in records x record size in bytes)}}{180}$$

2. For a keyfile, the following calculation yields the maximum keyfile size:

   A. IS = FS/NK
   B. RS = (((IS/32) + 27)/NK) + 3
   C. KS = IS + (larger of (IS/4) and RS)

   (IS = Index area size
   FS = maximum data file size
   NK = number of keys per sector: 5,7,11 or 22
   RS = Rough table size
   KS = maximum keyfile size)

Note that the maximum file size must be supplied through the appropriate field of the FPB at file creation time. The sizes of individual areas can be adjusted (in integral multiples of the original blocksize) at allocation time to minimize disk fragmentation.

Area allocation for dual pack files preferentially takes place on the pack referenced by the volume-id field of the FPB. If no room is available on that pack, an attempt is made to allocate the area on the other disk.

If no disk space can be found to allocate an area, the task requesting the area is suspended pending operator intervention. In the case of files currently on one pack, the operator may AD a second disk to make the file a dual pack file. Refer to table 2-9.

Table 2-9. Block within the Pseudo Pack Identifier Table

Byte	Contents	Size	Data Code
0-6	Pseudo pack identifier	7	A
7	Pseudo pack tag	1	B
8	Logical unit number	1	B
9-179	19 more 9 byte entries	171	–

Notes :

The pseudo pack tags in the range 01-#1F are reserved for PPIT entries of the component disks and 20 is reserved for the pseudo system pack.

The logical unit number is either that of the physical unit to which the pseudo pack is confined (restricted pseudo pack), or FF (extendable pseudo pack).

Pseudo packs are not implemented on the B 90 system.

TAPE PHYSICAL CHARACTERISTICS

A physical record on cassette can be between 2 and 256 bytes long. (On tape the upper and lower limits are 18 and 8192 respectively). A block must also conform to this restriction and therefore must end within the first physical record.

TAPE LOGICAL CHARACTERISTICS

Labelled files conform to Burroughs standard format illustrated in table 2-10. However, by setting the appropriate bit in the flags field within the corresponding FPB, tape files can be declared as unlabelled. An 'AD' SCL command would be necessary to inform the MCP of which unit the file is assigned to.

Table 2-10. Burroughs Standard Label Format

Byte	Contents	Related notes
0	Blank	
1-8	'LABEL– –0' where – is a blank	
9-15	Multifile id or '0000000' for single file cassettes	
16	Zero	
17-23	File id	
24-26	Reel number within a magnetic tape file	
27-31	Date written (creation date YYDDD)	
32-33	Cycle number: used to distinguish multi runs of the program	
34-38	Purge date (YYDDD)	1
39	Sentinel	2
40-44	Block count (on ending label only)	
45-51	Record count (on ending label only)	
52	Zero	
53-57	External magnetic tape library reel number	
58-62	Creation system	
63-70	Block size	
71-78	Record size	
79	Reserved	

Related Notes :

1. A write-enabled tape containing a valid label with an expired purge date is recognized as scratch, and one containing no label is recognized as unlabelled.
2. This byte is set to zero to denote the End of File (EOF) label and is set to one to denote the End of Reel (EOR) label which is affixed by the MCP when the cassette is full. The operator is then requested to supply the next reel for the remainder of the file.

Each tape file consists of the following components :

1. Beginning of file label (one physical record)
2. Tape mark
3. A set of consecutive physical records. File data, separated by inter physical record gaps
4. Tape mark
5. Ending file label (one physical record)

More than one file can be stored on a single tape. The last file can overflow onto another reel(s).

An unlabelled file omits items 1, 2 and 5. In addition, more than one unlabelled file can be stored on a single tape, but in this case, no overflow onto other reel(s) is permitted.

LINE PRINTER MEDIA

Line printer files can be labelled or unlabelled by the appropriate setting of a bit in the flags field within the associated FPB. Labels, if any, consist of '?DATA < file-name >' and '?END < file-name >' on otherwise blank pages.

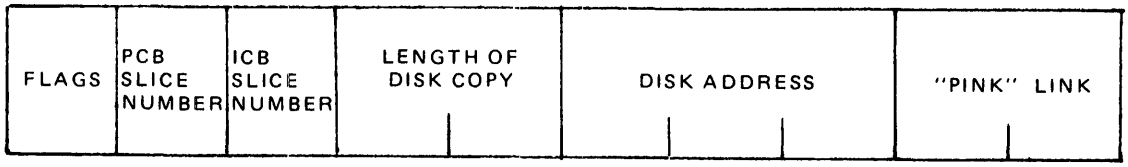
CONSOLE-PRINTER MEDIA

Console-printer files can be labelled or unlabelled by the appropriate setting of a bit in the flags field within the associated FPB. Labels, if any, consist of '?DATA < file-name > ' and '?END < file-name >' on otherwise blank pages.

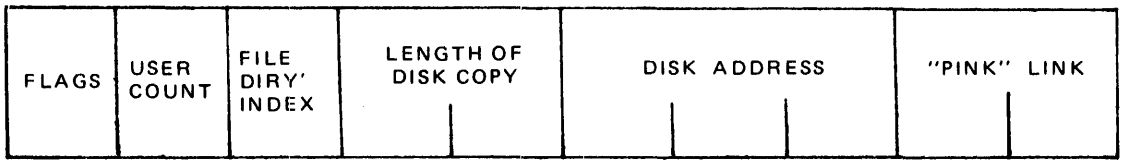
PUNCHED CARDS MEDIA

All card decks are labelled. The first card contains '?DATA – < file-name > – # # – – – # ' and the last card contains '?END – # – – – #' . The format of these labels is fixed; that is, '?DATA' and '?END' occurs in the first five columns of the card, with characters filling the remainder of the card. Note that – indicates a blank card column.

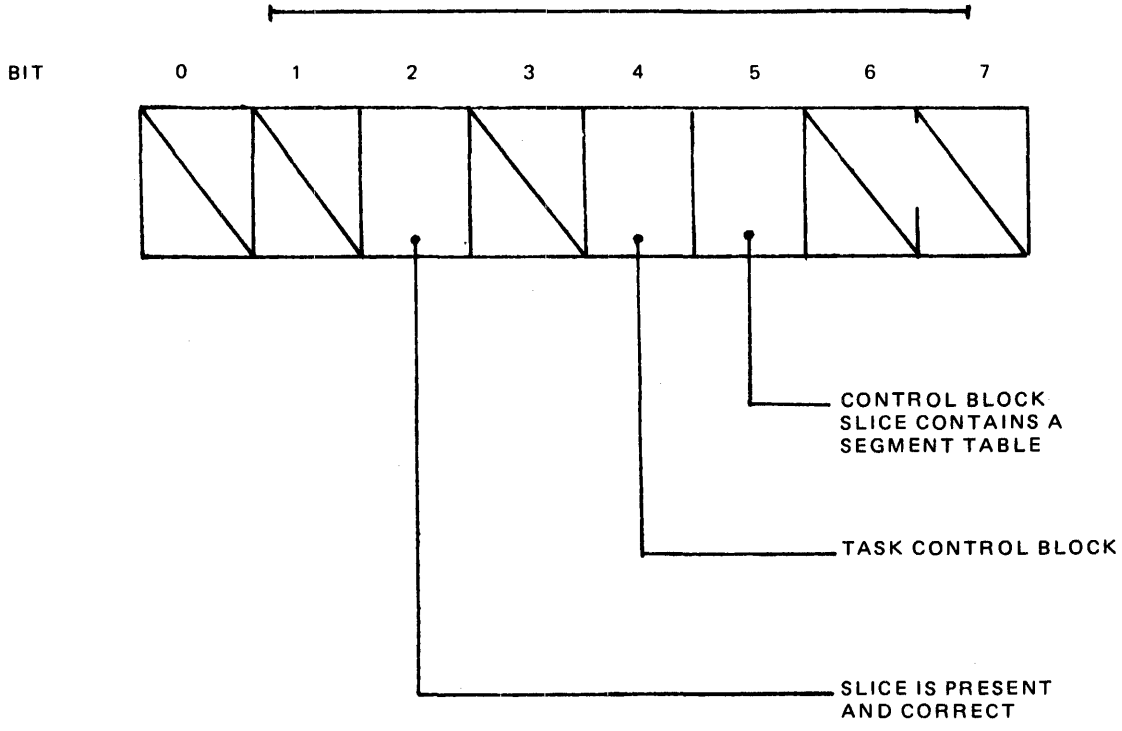
APPENDIX A
FIELD FORMATS



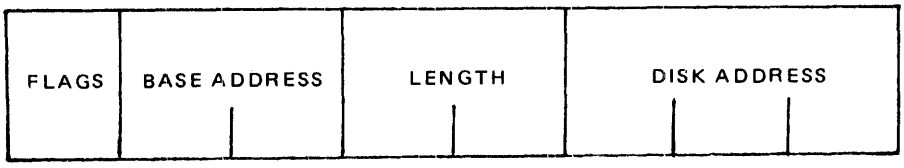
TCB SLICE DESCRIPTOR



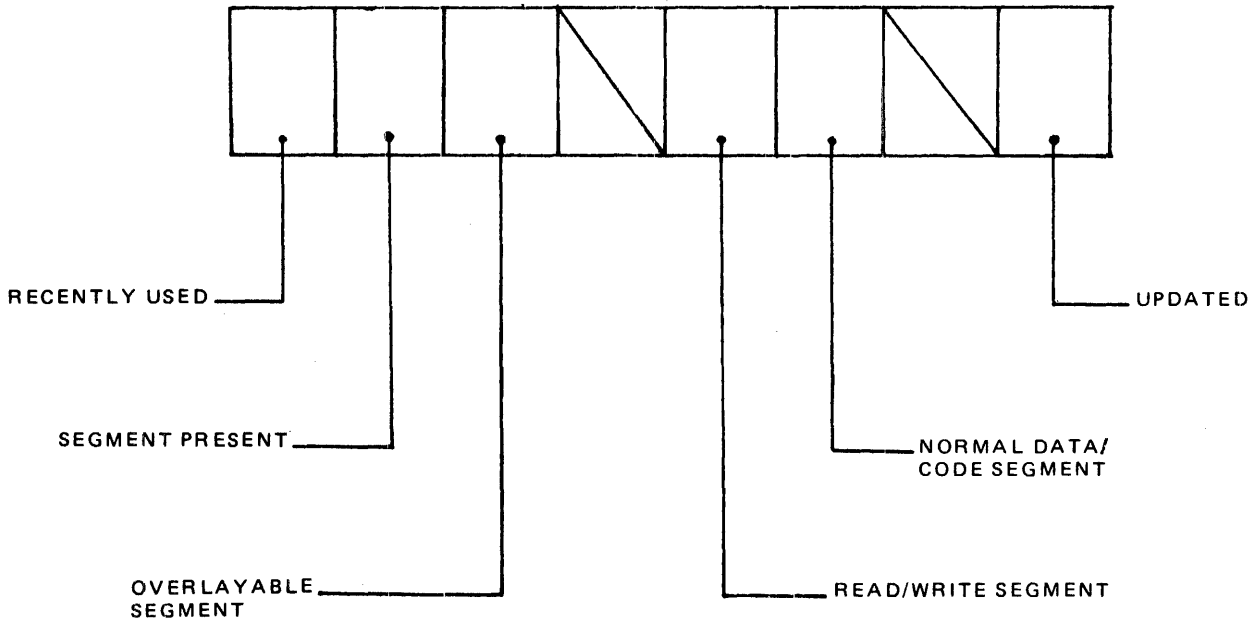
CCB SLICE DESCRIPTOR



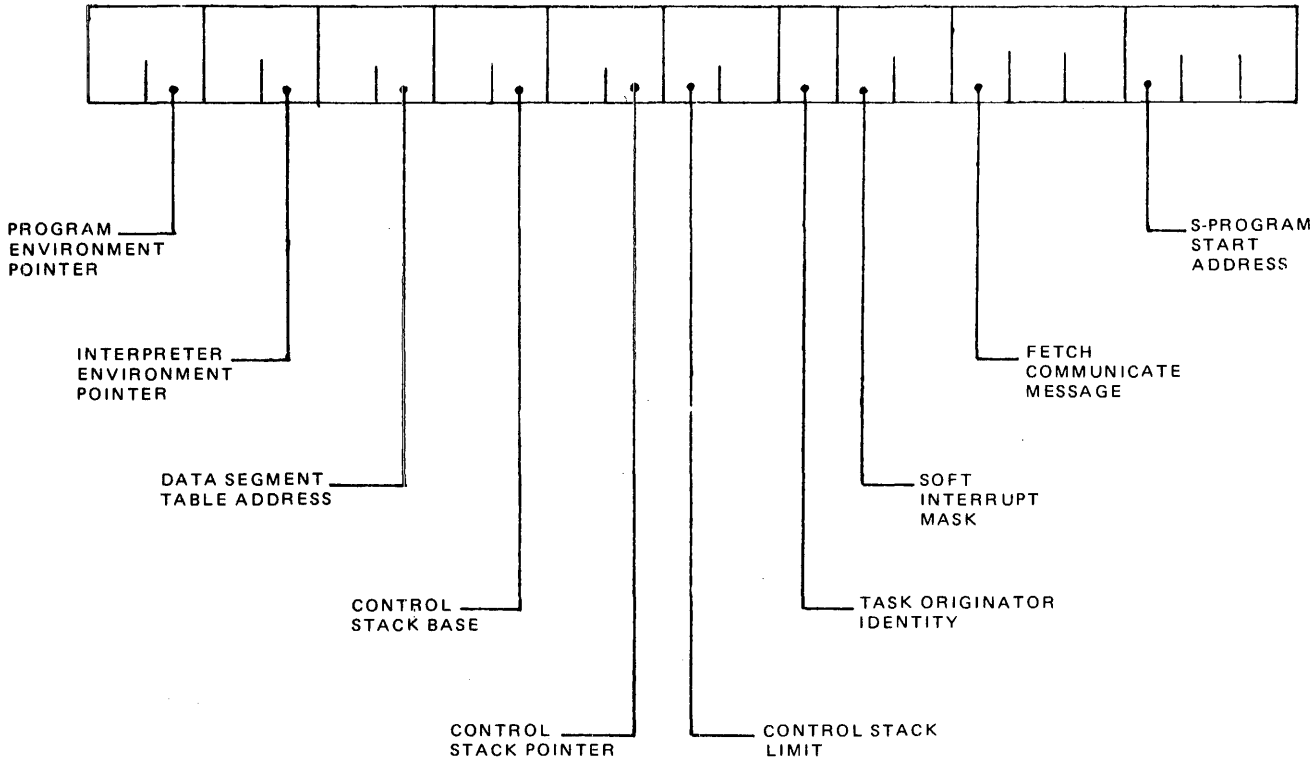
SLICE DESCRIPTOR FLAGS



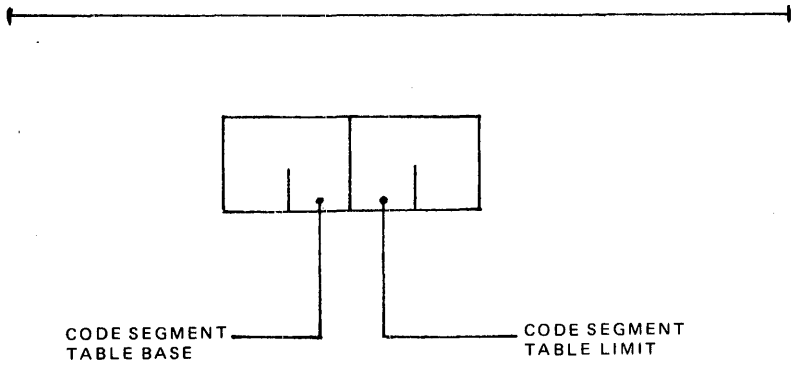
SEGMENT DESCRIPTOR



SEGMENT DESCRIPTOR FLAGS



TCB FIXED POINTERS



CCB FIXED POINTERS

Documentation Evaluation Form

Title: Computer Management Systems (CMS)
Master Control Program (MCP) Reference Manual

Form No: 2007555
Date: May 1982

Burroughs Machines Ltd is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Documentation Dept., TIO - Europe
Burroughs Machines, Ltd.
1 Tollpark Place, Wardpark East
Cumbernauld
Glasgow, G68 OLN
Scotland