NASA CONTRACTOR REPORT

MAMOS:

A MONITOR SYSTEM UNDER *IBSYS* FOR THE IBM 7090/7094

by Alfred E. Beam

Prepared by
UNIVERSITY OF MARYLAND
College Park, Md.
for

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • MAY 1966

NASA CR-488

MAMOS:

A MONITOR SYSTEM UNDER IBSYS FOR THE IBM 7090/7094

By Alfred E. Beam

Prepared under Grant No. NsG-398 by
UNIVERSITY OF MARYLAND
College Park, Md.

for

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

For sale by the Clearinghouse for Federal Scientific and Technical Information
Springfield, Virginia 22151 – Price $7.00

Abstract

This report describes an operating system which operates on the IBM 7090/7094 under the IBSYS or DC-IBSYS Monitor. The system processes jobs written in the MAD language, ALGOL language, FORTRAN language, and UMAP language. The processors of the system are among the fastest currently available. A very extensive library of programs is also provided. The system is especially useful for processing student jobs.

TABLE OF CONTENTS

TABLE OF CONTENTS - Continued

Acknowledgements

In the past months there have been many man hours expended in the preparation of the completed version of MAMOS. I would like to express my deep appreciation to the many people who have so freely contributed their time and effort in the preparation of MAMOS and without whose help, completion of the system would not have been possible.

Much credit for the system must be given to the staff of the University of Michigan Computing Center, for without the many components of the Michigan Executive System which were adapted for the MAMOS monitor, MAMOS would not be worthy of being called a system. I am very grateful to Professor Bernard Galler of the University of Michigan for making the entire Michigan Executive System and write-ups available to the Computer Science Center.

Many thanks are due to the staff of the University of Illinois Digital Computer Laboratory for their contribution of the ALCOR-Illinois ALGOL compiler which was made a component of MAMOS.

I would like to express my thanks to Messrs. John Bielec, George Lindamood, William Cleveland, Howard Wactlar, and Dr. Earl Schweppe who were very helpful in checking various parts of the system through their early use of MAMOS. Special thanks go to Mr. Robert Herbold for his excellent help in finding some very elusive bugs, and to Mr. John Montague for writing some subroutines and several test programs for the general check-out of the system. Also to Mr. Gunter Meyer for writing some subroutines.

Early use of MAMOS in courses taught by Dr. Earl Schweppe, Dr. Howard Tompkins, and the faculty of the Electrical Engineering Department greatly aided in checking out the system.

It is a pleasure to acknowledge also the valuable assistance rendered to MAMOS by a number of other installations. In particular to Mr. William Cahill, Head of the Mathematics and Computing Branch of the Theoretical Division of the Goddard Space Flight Center for the use of his installation in checking MAMOS under the Direct Coupled System (DC-IBSYS). Also to Messrs. Nate Dillard and Sam Wax of the above installation for their aid in getting MAMOS edited into their system. To Dr. Robert F. Rosin of the Yale Computing Center for early use of MAMOS under DC-IBSYS and detection of some errors in the system.

Finally, many thanks to Mr. Jack Otley for his general assistance and in particular for getting together the description of the MAMOS library; and to Mrs. Stella Tobin, Miss Alexandra Sieg, and Miss Carol Fung for typing the write-up.

1 The MAMOS Monitor And JOB Definition

1.1 Introduction

This write-up describes the 7090/7094 MAMOS Monitor. MAMOS is a sub-monitor in that it operates under the IBM Basic Monitor (IBSYS).

MAMOS operates under 7090/7094 IBSYS which has at least eight IBM 729 tape units, or under IBSYS for IBM 704X to 709Y direct coupled systems.

MAMOS processes programs written in the MAD language, the FORTRAN II language, ALGOL language, and the UMAP language. UMAP is very similar to the FAP language under the FORTRAN II Monitor. Other language translators will be added to MAMOS in the future.

Chapter 3 of this write-up contains a MAMOS description which is written for the general user. There are several references to section 1.2 in Chapter 3, and except for section 1.2 the general user may ignore the first 2 chapters of this write-up.

Chapter 2 is written mainly for the systems programmer.

1.2 Sub-System Selection and Job Identification

A deck of cards [DECK] is defined here as a card deck arranged in such a manner that when operated upon by MAMOS, it produces the desired result for the user.

A job deck [JOB] is defined as one or more IBSYS control cards followed by a [DECK]. The IBSYS control cards are described in IBM 7090/7094 Operating Systems Basic Monitor (IBSYS).

Individual installations have several options in their use of the IBSYS control cards and it is impossible to describe, in general, the specific IBSYS cards required for a [JOB]. Hence a [JOB] is now defined for MAMOS as it operates under IBSYS at the University of Maryland Computer Science Center.

A job deck [JOB] is defined as two IBSYS control cards followed by [DECK] as follows.

```
$EXECUTE        MAMOS
$ID   name*task*options$any comment
   [DECK]
```

$EXECUTE is punched in card columns 1-8 and the system name MAMOS is punched starting in card column 16. This card is used by IBSYS to locate on the system tape, call in, and relinquish control to MAMOS.

The second card above is the job (or identification) card, and it serves two purposes.

1) It is a signal for the beginning of a job.

2) It provides information for accounting.

The job card has $ID punched in card columns 1-3 and the name, task, and optional fields are punched in card columns 7-66. The fields are separated by an asterisk (*).

The dollar ($) character and the asterisk (*) character are not allowed within any of the fields. Any desired comment may follow the dollar sign which terminates the fields.

The name and task fields are described as follows.

i) name field: This field consists of the users last name followed by any other identifying characters (except $ and *) the user may wish to use. The name field can be from 6 to 18 characters in length.

ii) task field: This field consists of 10 characters having the form xxx/yy/zzz where

xxx is the department identification.
yy is the year the task was established.
zzz is the task serial number.

There are 6 optional fields which may be specified:

1) time field: This field consists of a decimal integer followed by by the character S (for seconds) or the character M (for minutes). The specified time is the maximum time which the program will be allowed to run during execution. If the time specified is exceeded, then execution is terminated and processing of the next job is begun.

2) print field: This field consists of a decimal integer followed by the character P (for pages). The specified number of pages is the maximum number of pages allowed to be written on the output tape during execution.

3) punch field: This field consists of a decimal integer followed by the character C (for cards). The specified number of cards is the maximum number of cards allowed to be written on the punch tape during execution.

4) dump field: This field consists of two decimal integers separated by a slash and the second integer is followed by the character D (for dump). The two integers may be any values between 0 and 32767, and they specify lower and upper limits of an IBSYS dump which will be taken if certain error conditions occur.

5) tapes field: This field consists of a word beginning with the character T (for tape). This field causes an on-line message to be printed for the operators. The message is followed by a pause so that the operators can save any tapes the user may have requested to be saved at the beginning of his job. The message is printed and the pause occurs just before the next job is signed on. This provides a safe-guard against missing a programmed pause and the next job over-writing save tapes.

6) punch delete field: This field consists of a word beginning with the character N (for NODECK). If this field is given, it specifies that the information written on the punch tape during the job is not desired as part of the job output.

The order of the 6 optional fields is not important. They are identified strictly by the first non-decimal character following an asterisk. Hence rather than the characters S, M, P, C, D, T, and N; the words SECONDS, MINUTES, PAGES, CARDS, DUMP, TAPES, and NODECK could be used as long as the desired specifications do not go beyond column 66 on the $ID card. Blanks are ignored.

The dump option on the $ID card will cause a dump to be given in case of excessive execution time or program hang-up.

If the estimates are not given by the user, then up to 30 seconds of execution time, 10 pages of printed output, and 20 punched cards are given. These numbers may change in the future.

Examples of $ID Cards

```
$ID   MARSIE,J.Q.*XXX/YY/ZZZ*20S*15P*NODECK*4096/8192DMP$T1
```

The above card specifies that the maximum execution time is to be 20 seconds and the maximum output during execution will be 15 pages (at 60 lines/ page). If the execution lasts more than 20 seconds then a core dump from 4096 to 8192 is to be given. No punched output is given.

```
$ID   DOTES  * XXX/YY/ZZZ*3M*10CRD *TAPES $T2
```

The above card specifies that the maximum execution time is to be 3 minutes, and that any punched output during compilation/assembly plus up to 10 cards during execution is to be given as part of the job output. Also there is to be a tape-save message and pause just before signing on the next job.

When the user submits a job to the Computer Science Center for processing, he also fills out a Run Submittal Card and the dispatcher prefaces the job deck with this card. The Run Submittal Card is prepunched with information which is used by the card-to-tape program when it prepares the job on tape for processing by the 7090/7094. The card-to-tape program, on recognizing the Run Submittal Card, inserts an end of file and the IBSYS control Card $JOB on tape in front of the job deck. After the job is completely processed, the Run Submittal Card is retained by the Computer Science Center.

If the user wishes to submit 2 or more job decks as a single deck he should separate the decks with an end of file card and a $JOB card. The end of file card should not have punching other than 7 and 8 punches in column 1.

2 MAMOS System Operation And Installation Options

2.1 Introduction

MAMOS is a very efficient job processor when properly placed on IBSYS Operating System Tapes. There are several options available to the individual installation as to where the system should reside on the operating system tapes. The speed of operation for tape systems is highly dependent on the placement of MAMOS.

No change is necessary to make the distributed version of MAMOS operate under direct coupled IBSYS (DC-IBSYS).

Simple parameters are provided to allow MAMOS to operate under various situations.

2.2 The MAMOS Distribution Tape

MAMOS is distributed as several files, taking almost a full reel of tape. The files are as follows.

FILE 1 : Version 12 IBSYS and DUMP records, almost as distributed by IBM. Tapes are all set for high density. IBEDT is not available on the tape.

FILE 2-4: MAMOS as an operating sub-monitor under IBSYS. The contents of the 3 files are described below.

FILE 5 : A MAMOS job which edits the MAMOS library. The binary decks as data are included in the job.

FILE 6 : A MAMOS job which produces absolute binary cards for the IBSYS MADTRAN record.

FILE 7 : The symbolic cards for the MAMOS system. The file was produced by FAP update, hence blocked records consist of up to 16 cards per record. The file may be used as an update input tape, or it may be used as an input (SYSIN1) tape to MAMOS.

FILE 8 : The two IBSYS control cards: $IBSYS and $STOP .

2.3 MAMOS Operating Arrangement

As distributed, MAMOS (as an operating sub-monitor) consists of 3 consecutive files of the IBSYS Operating Systems Tape. The 3 files are organized as follows. The first word in the list gives the name of the record which is used in editing.

```
MAMOS  :   Input/output supervisor.
End of file
MADML1 :   MAMOS Monitor, Loader, and Accounting programs.
MADCPL :   MAD Compiler.
UMAPAS :   UMAP Assembler.
ALD001 :   ALGOL Compiler - part 1.
ALD002 :   ALGOL Compiler - part 2.
ALD003 :   ALGOL Compiler - part 3.
ALD004 :   ALGOL Compiler - part 4.
ALD005 :   Short dummy record.
LIBHED :   Library header record.
The MAMOS library consisting of many records.
End of file
COPIES :   Record used to make copies of job output.
MADTRN :   MADTRAN Translator.
End of file
```

All of the above programs are written in UMAP except MADTRN and a few of the library routines, which are written in the MAD language.

If the system is assembled, the binary output must be rearranged a little before doing an edit to produce MAMOS as an operating sub-monitor.

1) The binary cards labeled COPIES which are part of the Monitor Assembly must be placed in proper position for the COPIES record.

2) The relocatable binary decks of MADTRAN must be run in a special program to punch out absolute binary for the IBSYS record MADTRN.

3) The low core library subroutine should be placed at the end of the library, and the library should be rearranged so that the most frequently used routines are at the beginning of the deck.

4) A library edit run must be made with the relocatable subroutine decks as data. The run produces a file on SYSCK2 which is suitable for a *DUP into the IBSYS Operating System.

It is hoped that the above steps will not be necessary for most installations since the operating system as distributed already has a good library arrangement.

Relocatable Program Decks For The Library

There are more than 100 subroutines in the library. File 5 of the distribution tape contains the relocatable binary decks for the library as data for the library editor. The editor produces a file of records which is then edited into IBSYS. The form of file 5 is as follows.

```
$JOB
$EXECUTE        MAMOS
$ID
$ EXECUTE
  [binary program deck for the library editor]
$ DATA
  [relocatable binary program decks for the library]
7/8   end of file card
```

As pointed out before, the arrangement of the decks is different from the order of the symbolic programs for the library.

If file 5 is run as a job, the edited library will be put on SYSCK2 as one file. The file is of the form required by the MAMOS loader, and also in a form suitable for editing into most IBSYS Operating Systems by means of the *DUP control card. There may be some DC-IBSYS systems which will not accept the library in the above form. In the case of the DC-IBSYS systems which require the first word of a library record not to be an IOCP, a simple copy and modify will produce an acceptable format. The library file which is in the third file of the distribution tape has already gone through such a transformation and it is hoped that the format is acceptable to all IBSYS systems. The format of the first word of each record is IORT 4096,0,N where N is the number of words in the record. The second word of the record is a BCD name. The MAMOS loader ignores the first two words of a library record.

The UMAP program which was used to produce the altered file is given below, and provides an example of the use of the basic I/O routine of MAMOS. The program assumes that tape mounting and saving instructions are given elsewhere.

```
$EXECUTE        MAMOS
$ID
$ ASSEMBLE, EXECUTE
        TITLE       FIX LIB FOR D-C IBSYS
READ    STZ     EOFS
        CALL    RDSBIN
        TIX     EOR,0,10            IN ON SYSCK2
        TIX     LIN,7,0
        TIX     EOF,0,0
        ZET     EOFS
        TRA     SYS
        TRA     WRITE
*
 LIN    IOBT    A,0,-1              READING COMMAND
*
 EOR    ALS     18                  USERS EOR TRAP TIME
        STD     LOUT                ROUTINE TO SET UP
        STD     WD1                 THE RECORD LENGTH.
        TRA     1,4
*
 EOF    STL     EOFS                USERS EOF TRAP TIME
        TRA     1,4                 ROUTINE.
*
 LOUT   IOBT    A,0,**              **= RECORD LENGTH
*
 WD1    TXH     4096,0,**           **= RECORD LENGTH
*
 EOFS   PZE     0
 A      BSS     20000               MAX. REC. LENGTH IS 200000
 WRITE  CLA     WD1
        STO     A
        CALL    WRSBIN
        TIX     0,0,11              OUT ON SYSCK1
        TIX     LOUT,7,0            7 TO FORCE WAIT
        TIX     0,0,0
        TRA     READ
*
 SYS    CALL    WEFTAP              PUT EOF ON SYSCK1
        TIX     0,0,11
        CALL    RUNTAP              UNLOAD SYSCK1
        TIX     0,0,11
        CALL    RUNTAP              UNLOAD SYSCK2
        TIX     0,0,10
        CALL    SYSTEM              END THE JOB
        END
```

The Job To Produce An IBSYS Record For MADTRAN

File 6 of the distribution tape consists of a job which when run, produces an absolute binary deck for MADTRAN on SYSPP1. The output deck is in a form suitable for editing into IBSYS as the MADTRN record. The MADTRN record in file 4 of the distribution tape was produced by running this job. The form of file 6 is as follows.

```
$JOB
$EXECUTE      MAMOS
$ID
$ ASSEMBLE, EXECUTE
  [Symbolic main program for MADTRAN]
  [Relocatable binary programs for MADTRAN]

$ DATA
7/8     end of file
```

The main program above has in it a one-time only program which punches out the absolute binary deck.

There are several ways to accomplish the above, but this method usually produces less binary cards and hence a shorter system record. The symbolic program above could be modified to produce absolute decks for other translators which are written as relocatable programs.

The Symbolic Programs For The MAMOS System

File 7 of the distribution tape contains all of the current MAMOS system in symbolic form. The file was produced by FAP update, and cards are blocked up to 16 cards per record. The entire system may be assembled by making a copy of files 7 and 8 on SYSIN1 and making a run, using the first 4 files of the distribution tape as the IBSYS Operating System. Such an assembly will produce much output on SYSOU1 and about 2 boxes of binary cards on SYSPP1.

A basic FAP update deck is listed below for those who wish to update and assemble the system. It is assumed that the FAP update input tape is positioned in front of file 7 on SYSCK2 and a scratch (for FAP update output) is ready on SYSCK1. It is further assumed that FORTRAN II logical numbers 9 and 10 refer to SYSCK1 and SYSCK2 respectively.

File 7 consists of 22 programs, followed by the library. The programs, in order, are as follows.

Program		Program	
1)	MAMOS	13)	ALD005 (a dummy record)
2)	MONITOR and COPIES	14)	LIBHED (small record)
3)	LOADER	15)	DUMLIB (small record)
4)	DOGTAG	16)	MADTRAN (main program)
5)	MAD 1	17)	MADTRAN (written in MAD)
6)	MAD 2	18)	MADTRAN (written in MAD)
7)	MAD 3	19)	MADTRAN (written in MAD)
8)	UMAP	20)	MADTRAN (written in MAD)
9)	ALGOL 1	21)	MADTRAN (written in MAD)
10)	ALGOL 2	22)	LIBRARY WRITER
11)	ALGOL 3	23)	LIBRARY (about 150 programs)
12)	ALGOL 4		

Basic UPDATE For MAMOS

```
1       8         16
$EXECUTE          FORTRAN                                                    C
$ID    NAME*001/65/003                      $                              0
*                 MOUNT TAPE XXXX ON FORTRAN II TAPE 10 = SYSCK2             L
*                 MOUNT SCRATCH ON FORTRAN II LOGICAL TAPE 9 = SYSCK1        U
*                 AFTER THE UPDATE USE SYSCK1 AS SYSIN1 TO PRODUCE A NEW     M
*                 ASSEMBLY OF MAMOS                                          N
*
*                                                                            7
*       PAUSE                                                                3
*       FAP
         UPDATE   10,9,,D
         REWIND   9
         REWIND   10
         ENDUP
*       FAP
         UPDATE   10,9,,D
         END                    END OF MADTRAN DRIVER                        M2499990
         ENDUP
*       FAP
         UPDATE   10,9,U,D
                   END OF FUNCTION                                          M2522980
         ENDUP
*       FAP
         UPDATE   10,9,,D
         END                                                                S2802340
         ENDUP
*       FAP
         UPDATE   10,9,U,D
                    END OF FUNCTION                                         S2902260
         ENDUP
*       FAP
         UPDATE   10,9,,D
         END            END OF REGRESSION PROGRAM                           T3925700
         ENDUP
*       FAP
         UPDATE   10,9,U,D
                    END OF FUNCTION                                         T4000890
         ENDUP
*       FAP
         UPDATE   10,9,,D
$$                   END OF MAMOS LIBRARY                                  XXX00000
         ENDFIL   9
$IBSYS
$*                END OF MAMOS ASSEMBLY*******************************
$PAUSE
         ENDFIL   9
         REWIND   9
         UNLOAD   10
         ENDUP
7/8             END OF FILE
$IBSYS
$*              SAVE TAPE XXXX, MAKE SCRATCH TAPE SYSIN1, AND START.
$PAUSE
```

The above update is basic to the MAMOS symbolics because of the blocking. Assemblies are much faster when the system is updated in blocked form. There are several MAD programs in the system and this is the reason for switching from blocked to un-blocked output.

UMAP is the only translator under MAMOS which, at present, accepts blocked input.

It should be mentioned that the DOGTAG program (fourth program of file 7) is not used at all in the distributed version of MAMOS and it may be deleted. DOGTAG was included because it is the accounting program used at the University of Maryland Computer Science Center, but its function is entirely deleted by parameter. DOGTAG, when used, is a floating program which is inserted into any available space in every sub-monitor under IBSYS. It then becomes a sub-monitor's responsibility to insure that DOGTAG is in core whenever a $JOB or $ID card is read. After processing $JOB and/or $ID the particular copy of DOGTAG can be wiped out. The advantages of this method are:

1) Many functions can be carried out without taking core away from the user.

2) The permanent core requirements for a partial SYSIDR routine, time trap routine, etc., are small and the required core can be merged with IBSYS-IOEX in the first 2K of core.

2.4 Options And Assembly Parameters

There are several options in MAMOS which may be effected by assembly parameter or in some cases by simple patches.

Arrangement of MAMOS On The Operating System Tapes

As distributed, MAMOS is rather inefficient for small job processing, since the library must be found and repositioned for each execution. Operation is greatly improved by placing the MAMOS library on a second library tape (SYSLB2). A further improvement is obtained when the MAMOS library is made the first file on SYSLB2.

The placing of the MAMOS library on SYSLB2 may be accomplished by an IBSYS edit and changing one parameter in the MAMOS record. The parameter (called LIBDIF) may be reset either by assembly or by patching. As distributed, LIBDIF (at octal location 3727) is zero. If the library is moved to SYSLB2 then LIBDIF should have the form

```
LIBDIF     TIX     F,0,8
```

where F is the number of files in front of the MAMOS library on SYSLB2 which is logical tape 8. Logical tape 8 may be redefined to be any other SYSLBi by changing the master I/O unit table entry for logical tape 8.

MAMOS can operate from any of the four library tapes. Logical tape 1=SYSLB1 in the master I/O unit table is automatically adjusted to be the actual SYSLBi on which MAMOS resides.

An IBSYS edit to place the MAMOS library as the first file on SYSLB2 could be as follows.

```
     *MODIFY   MAMOS
3727 *OCT      200010000000
     *AFTER    LIBHED
     *DUP      SYSLB1,SYSLB2,1
     *INSERT   DUMLIB
4061 *OCT      246444433122
4060 *OCT      000000000014
 100 *OCT      002000000115
     *INSERT   FILEMK
```

In the above, it is assumed that SYSLB2 is attached properly in IBSYS and that SYSLB1, and SYSLB2 are properly positioned prior to reaching the above edit cards. The DUMLIB record which is inserted above could be the DUMLIB assembly of the symbolic program mentioned previously.

Blocked Output Option For Translators

As distributed, all printed output on SYSOU1 is single record buffered, one print line per record. A considerable improvement in operation is achieved if the peripheral processor can handle blocked output. In MAMOS, the option to allow blocked output for translators is provided. The option is activated by reassembly of the MAMOS record with the parameter OU1BUF (at octal location 3723) set non-zero. Patching of OU1BUF is also possible.

If blocking is activated, then a physical record will consist of up to 660 characters, with each print line (except the last) followed by a record mark (octal 72 in 7090/7094).

Processor Calls To SYSIDR

As distributed, various processors of MAMOS make calls to SYSIDR. The calling sequence is as follows.

```
        TSX     SYSIDR,4
        PON     0,i,0
        Return
```

where i=1 to sign on a compiler.
2 to sign on an assembler.
3 to sign on a loader.
4 to sign on execution.
5 to sign on end of job process (including dumps).
6 to sign on between segment processing.

The installation which has an accounting routine that is unable to handle the above calls should disable them by reassembling the MAMOS record with the parameter PROCON defined as

```
PROCON  EQU   0
```

rather than its distributed definition as

```
PROCON  EQU   1 .
```

Requirement Of $ID Cards

As distributed, a $ID card is required for each job. To delete this requirement, reassemble the Monitor record with the definition of NOIDOK being

```
NOIDOK  EQU   0
```

rather than

```
NOIDOK  EQU   1 .
```

Option Of Putting The $ID Card on SYSPP1

As distributed, MAMOS puts the $ID card on the punch tape (SYSPP1) for identification of the punched output. To delete this, reassemble the Monitor record with the definition of PNCHID being

```
PNCHID  EQU  0
```

rather than

```
PNCHID  EQU  1
```

Option Of Putting The $JOB Card On SYSPP1

If desired, MAMOS will put the $JOB card (if any) on SYSPP1 for identification of punched output. To activate this option, define PNCHJB in the Monitor as 1 rather than 0 and reassemble the Monitor. If the $JOB card is put on SYSPP1, it is done just before going to the IBSYS entry SYSRPT.

Options When There Is No Accounting Routine (SYSIDR)

There are several parameters defined in the Monitor record in case an installation does not have an accounting routine. If any of these parameters are changed then the Monitor record should be reassembled. These options have no meaning for those installations which have their own accounting routines. All of the parameters are defined by the pseudo-operation EQU and the first possible value of each symbol is the value in the distributed version of MAMOS.

OFFJOB = 2 if MAMOS is to print $JOB cards on SYSOU1 on a new page.

= 1 if MAMOS is to print $JOB cards on SYSOU1 under double space control.

= 0 if MAMOS is not to print $JOB cards on SYSOU1.

ONJOB = 2 if MAMOS is to print $JOB cards on-line on a new page.

= 1 if MAMOS is to print $JOB cards on-line under single space control.

= 0 if MAMOS is not to print $JOB cards on-line.

OFFID is defined in the same manner as OFFJOB except it applies to $ID cards.

ONID is defined in the same manner as ONJOB except it applies to $ID cards.

Separation Of Jobs On SYSOU1

As distributed, MAMOS puts an end of file in front of each job's output on SYSOU1. To delete these file marks, redefine JOBEOF as

```
JOBEOF  EQU  0
```

rather than

```
JOBEOF  EQU  1
```

and reassemble the Monitor record. The end of file (if JOBEOF = 1) is written just before going to the IBSYS entry SYSRPT.

If the above parameter change is made then the $ COPIES specification will not be effective, since copies of job output is made by back-spacing the output to the beginning of the job output.

It should be remembered that the COPIES record is obtained as part of the Monitor assembly, and if the Monitor is reassembled and storage is changed, then the binary cards for the COPIES record must be re-edited into IBSYS.

IBSYS Control Cards ($JOB, $EXECUTE, $ID)

It is possible to run MAMOS jobs headed by IBSYS cards as follows.

```
$JOB
$EXECUTE      MAMOS
$ID
 [DECK]
```

or

```
$EXECUTE      MAMOS
$ID
 [DECK]
```

or

```
$JOB
$EXECUTE      MAMOS
 [DECK]
```

There should never be trouble when the first two arrangements are used. The third arrangement will give trouble on DC-IBSYS when UNKNOWN jobs are run, because results from a previous TEACHER job will be lost.

When MAMOS reads a $JOB it does all things described in the IBSYS manual except to kill executions of sub-jobs when a previous sub-job fails. The reason for not including this feature was to allow more flexibility in setting up student jobs.

Reservation Of Upper Memory For Accounting Routines

As distributed, MAMOS uses all of core above $(3721)_8$. There is a parameter, called SYEND, in several programs, and the parameter may be defined (by the EQU pseudo-op) as the number of cells (up to 500) which are to be reserved in high core for accounting purposes. SYEND is currently defined as zero, and appears in the Monitor, Loader, ALGOL-2, ALGOL-3, and A)PTRA programs. A)PTRA is a library program which is used during execution of ALGOL programs.

If SYEND is changed then the above programs must be reassembled. In loading relocatable programs for execution, the loader gets its value of SYEND from the IBSYS location SYSCOR.

The symbolic card labels of the cards containing SYEND EQU 0 are M0200670,M0300170,M1400890,M1600870, and T4700030.

3 MAMOS Monitor System Under IBSYS

3.1 Introduction

The 7090/7094 MAMOS Monitor System is a monitor system which operates under IBSYS.

MAMOS has several components from the University of Michigan Executive System for the IBM 7090 Computer (henceforth called MES). MES is one of the most advanced monitors available for a two channel tape system. The main disadvantage of MES is that it would be very hard to incorporate into the system, the recent sub-monitors which are now available under IBSYS.

MAMOS is an attempt to make available under IBSYS the most important parts of MES; namely the very fast compiler, the assembler, the loader, and the extensive library. The ALGOL Compiler is also under MAMOS. Other translators will be added to the MAMOS system in the future.

The object of this chapter is to provide necessary details and control card descriptions for programmers who will use various components of MAMOS. The chapter is presented in sections which may be referred to as the need arises.

Several examples of job decks are given in this chapter. In examples which include IBSYS control cards, these cards are the ones used at the University of Maryland Computer Science Center. The user at other installations should refer to section 1.2 for information on the required IBSYS control cards.

Programmers who are only interested in writing and running MAD programs may turn immediately to section 3.3 and ignore the rest of the chapter.

Programmers who are only interested in writing and running ALGOL programs may turn immediately to section 3.4.

3.2 MAMOS Control Cards and Their Functions

All MAMOS control cards have a dollar ($) sign in column 1 and the control specifications are punched in columns 2 through 64. If more than one specification is punched on a single control card they are separated by commas. The order in which specifications are punched is not important and blanks are ignored.

Columns 65-72 may contain identification (of which the last 3 columns are numeric) which will be used to identify binary cards produced by the system.

The specifications indicate to MAMOS what is to be done as the job is processed. The specifications are divided into two groups. The first group should appear at the beginning of the job deck to indicate over-all control of the job. The second group describes what is to be done with individual parts of the job deck.

First Group

EXECUTE	:	Indicates that the object deck resulting from processing of the job deck should be executed. If an error occurs during processing, no execution will take place.
COPIES(N)	:	Causes N-1 additional copies of the output to be produced.
DUMP	:	Indicates that in case of trouble during execution, a dump of the program and erasable storage is to be given.
FULL DUMP	:	Indicates that in case of trouble during execution, a dump of 0-4095, the program and erasable storage is to be given.
I/O DUMP	:	Indicates that a dump is to be given if an I/O error occurs during execution.
SUB TRACE	:	Indicates that during execution a print out should be given of all subroutine entries, except those subroutines called by library subroutines and those subroutines called implicitly by subscription in MAD.
REGRESSION	:	Indicates that the library subroutine for regression is to be automatically called in and control given to the subroutine.
TEACHER	:	Indicates that student jobs follow the job this specification appears in.
UNKNOWN	:	Indicates that the job is a student job and that the job is to be combined with a tape prepared by a previous teacher job.

Second Group

COMPILE MAD	:	Indicates that the MAD program which follows is to be compiled.
COMPILE ALGOL	:	Indicates that the ALGOL program which follows is to be compiled.
ASSEMBLE	:	Indicates that the UMAP program which follows is to be assembled.
MADTRAN	:	Indicates that the following FORTRAN II program is to be translated to MAD language and then the MAD program is to be compiled.
PRINT OBJECT	:	Indicates that the object program which results from the current MAD compilation is to be printed.
PUNCH OBJECT	:	Indicates that the object program generated by the current $COMPILE MAD, $COMPILE ALGOL, or $ASSEMBLE is to be punched in column-binary form.
BINARY	:	Indicates that only binary cards follow up to the $DATA card.
BINARY(N)	:	Indicates that the binary program is to be loaded from logical tape N after the execution tape is loaded. N may be 2,3,4,7, or 9.
DATA	:	Indicates that the information which follows is data. The $DATA must always precede the first data card if data is present.
CONDITIONAL	:	Indicates that the section of the job deck which follows is to be completely by-passed if the EXECUTE specification was not given or if execution was deleted by some error which occurred in an earlier compilation or assembly.
BREAK	:	Indicates that the program from the job beginning or a previous $BREAK up to this $BREAK is to be considered as a core load to be used in a PING PONG job. The program is written for later execution as a systems record on logical tape 2.
BREAK(N)	:	Same as $BREAK except the system record is written on logical tape N. N may be 2,4, or 9.
HALT	:	Indicates that the computer is to stop to allow the operator to take action(s) indicated on previous comments cards ($ in columns 1 and 2). The $HALT is effective only if preceded by comments cards, and should only be used if necessary.

Second Group (Continued)

COMMENTS : Comment cards have $ punched in both column 1 and 2. Any legal punching may appear in card columns 3-72.

Note on EXECUTE and DATA specifications

The $EXECUTE and $DATA specifications may give trouble if MAMOS is operating under DC-IBSYS, and either specification is punched starting in Column 2. A convention of leaving Column 2 blank for the above specifications is desirable for DC-IBSYS users.

Termination of execution

Execution is normally terminated in one of the following ways:

1) By trying to read more data than was supplied in the data deck.

2) By the MAD statement EXECUTE SYSTEM. or symbolic instruction CALL SYSTEM .

3) By the MAD statement EXECUTE ERROR. or symbolic instruction CALL ERROR .

Termination by 3) should be used if requested dumps are desired. Termination because of some error detected by the system is always through ERROR routine.

Job Deck

A job deck for MAMOS begins with two IBSYS Control cards. These control cards are described in section 1.2. The two cards serve to tell the IBSYS Monitor which subsystem (in this case MAMOS) is desired and to provide accounting information for the job.

Following the IBSYS control cards are MAMOS control cards, MAD programs, UMAP programs, ALGOL programs, object programs, and data cards.

A job deck is processed by MAMOS working its way through the deck, calling in translators whenever specifications indicate their need. Binary programs resulting from translations and from the job deck are stacked on logical tape 3.

When the $DATA specification or end of job deck is encountered, a check is made to see if execution is still legal. If execution is not legal because it was not requested or because of some error, then the job is terminated.

If execution is legal then the program which was stacked on logical tape 3 is loaded. Then the library is searched for any subroutines which are needed by the program, and control is passed to the program for execution.

3.3 MAD Under MAMOS

MAD stands for Michigan Algorithm Decoder. MAD is a computer program (based on ALGOL 58) which translates algebraic statements describing algorithms to the equivalent machine instructions. A description of MAD may be found in several publications. The reference which describes MAD as it works under MAMOS is the Michigan Algorithm Decoder, The University of Michigan Computing Center by B. Arden, B. Galler and R. Graham. The above authors also wrote the computer program for MAD.

MAMOS Control Cards

MAMOS control cards are identified by having a dollar ($) sign punched in column 1. Control specifications are punched on the control cards in columns 2 through 64. If more than one specification is punched on a single control card, they are separated by commas. The specifications indicate to MAMOS what is to be done as the job is processed.

Necessary Control Specifications

$ COMPILE MAD

Every MAD source deck must be immediately preceded by this specification which indicates to MAMOS that the MAD compiler is needed. Binary deck identification may appear in columns 65-72. Columns 70-72 must be numeric.

$ EXECUTE

The above specification must be used if execution of a program is desired.

$ DATA

If execution is desired and execution requires a data deck [DATA], then the data deck must be preceded by the $DATA specification.

Optional Control Specifications

$ PRINT OBJECT

The above specification is used if it is desired to have printed, the object program (in octal) which is generated by the MAD compiler. This option is effective only for the MAD source deck which immediately follows the specification.

$ PUNCH OBJECT

The above specification is used if it is desired to get as part of the output a binary deck of the object program generated by the MAD Compiler. This option is effective only for the MAD source deck which immediately follows the specification.

There are several other specifications available and these are described in other sections.

Composition of Simple MAD Job Decks

The most simple application of MAD under MAMOS would consist of a compilation of one MAD program. The job deck would be made up as follows:

```
$EXECUTE        MAMOS
$ID   NAME*TASK*OPTIONS$COMMENT
$COMPILE MAD
   [MAD program]
```

The output from the above job would consist of a listing of the program and any diagnostics detected by the MAD compiler.

If we replaced the $COMPILE MAD card above with the card

$EXECUTE, COMPILE MAD

and if the MAD program compiled correctly, then execution of the compiled program would be attempted. Output from the job would be the same as above, plus any output generated during execution. Note that 2 or more specifications may appear on the same card as long as they don't go beyond column 64. It is assumed above that no data is required for execution of the program.

Example

The following example of a MAD job deck uses all of the 5 specifications described above. The program reads the value X, computes $F(X)=\sqrt[7]{X}$, then prints X and F(X). These 3 operations continue until the data deck is all read in and then the job is automatically terminated by MAMOS.

Computation of F(X) is done by means of an external function so there will be two compilations by the MAD compiler. If there is an error in either compilation, then execution will not be permitted.

```
$EXECUTE       MAMOS
$ID   MARSIEDOTES*XXX/YY/ZZZ*5S*1P$COMMENT
$ COMPILE MAD,EXECUTE,DUMP                                    MAIN0001
$PRINT OBJECT, PUNCH OBJECT
          RMAIN PROGRAM WHICH READS DATA, CALLS A
          RFUNCTION, AND PRINTS RESULTS
 START     READ DATA X
           EXECUTE FUNCT.(X,FX)
           PRINT RESULTS X,FX
           TRANSFER TO START
           END OF PROGRAM
$ COMPILE MAD,PRINT OBJECT                                    FUNCT001
           EXTERNAL FUNCTION(Z,W)
           ENTRY TO FUNCT.
           W= SQRT.(Z)
           FUNCTION RETURN
           END OF FUNCTION
$DATA
  X= 1.0       *
    X =3.1415926    *
```

In the above example, the first two cards are IBSYS control cards which are necessary for any job run under MAMOS. The $ID card above specifies that execution of the program is not to be allowed to run more than 5 seconds, and that no more than 1 page of output will be generated during execution. See section 1.2 for a complete description of the $EXECUTE and $ID IBSYS control cards.

The third specifies that (1) A MAD program which follows is to be compiled, (2) Execution of the program is desired, (3) If there is an error detected during execution, a dump of the program is desired, (4) The binary cards produced by the MAD compilation are to be labeled in columns 73-80 with MAIN0001, MAIN0002,.... The label which is used on output binary decks must be punched in columns 65-72 of the specification card.

The fourth card specifies that an octal print out of the instructions produced by the MAD compilation is desired, and also a binary deck of the object program is to be given as part of the job output.

The next 7 cards make up the first MAD source deck which is to be compiled, and the $ specification which follows specifies that a MAD deck which follows is to be compiled and an octal print out of compiled instructions is to be given. Labeling is also specified in columns 65-72 but this is ineffective since the PUNCH OBJECT specification is not given.

The next 5 cards make up the second MAD source deck which is to be compiled, and the $DATA specification which follows specifies that execution, if no errors have been detected, is to begin.

The final two cards make up the data deck to be used during execution.

An outline is given below of the output produced when the above job deck is run.

Printed output

1) The $ID and sign-on time.
2) The first MAMOS specifications.
3) A list of the first MAD source deck, and any detected errors in the source deck.
4) A list of individual MAD statements and the instructions compiled for each statement. This output is given only if there were no errors detected in compilation.
5) The MAMOS specification card for the second MAD source deck is printed.
6) A list of the second MAD source deck, and any detected errors in the source deck.
7) A list of individual MAD statements and the instructions compiled for each statement. This output is given only if there were no errors detected during compilation.
8) The $DATA specification is printed.
9) If no errors were detected in compilations then execution output is printed consisting of 2 lines of X,F(X). Also, just previous to the execution results is a printed list of subroutines used and their octal origins in core.
10) The requested program dump if an error occurs during execution.
11) The $ID and sign-off time.

Punched output

1) The $ID card and a binary deck of the instructions compiled for the first MAD source deck. The binary deck is not given if an error is detected during compilation of the first MAD source deck.

It should be noted in the above example that the specifications EXECUTE and DUMP are effective for the entire job while the COMPILE MAD, PRINT OBJECT, and PUNCH OBJECT specifications only apply to the MAD source deck following the specifications. This is true also for the option of putting binary card labeling information in columns 65-72 of a specification card.

COMPILE MAD is the only necessary specification for any MAD compilation.

Note on EXECUTE and DATA specifications

The $EXECUTE and $DATA specifications may give trouble if MAMOS is operating under DC-IBSYS, and either specification is punched starting in Column 2. A convention of leaving Column 2 blank for the above specifications is desirable for DC-IBSYS users.

3.4 ALGOL Under MAMOS

ALGOL stands for Algorithmic Language.
ALGOL was first defined in 1958 (see the December 1958 issue of Communications of the Association for Computing Machinery) and a new description of ALGOL (called ALGOL-60) was published in 1960 (see the May 1960 issue of Communications of the Association for Computing Machinery). The Revised Report on the Algorithmic Language ALGOL-60 was published in the January 1964 issue of Communications of the Association for Computing Machinery.

There are several ALGOL translators in use and development for several machines.

The ALGOL translator which operates under MAMOS is the ALCOR - University of Illinois ALGOL-60 Translator. The translator was written for the IBM 7090/7094 by people of the University of Illinois and the ALCOR group in Europe. A User's Manual for the ALCOR - University of Illinois ALGOL-60 Translator has been written by E. L. Murphree, Jr. of the University of Illinois.

Programmers who know ALGOL-60 as described in the Revised Report on the Algorithmic Language ALGOL-60 should, with the aid of this section, be able to write ALGOL programs and have them translated and executed under MAMOS.

Restrictions on ALGOL programs

1) The ALGOL translator produces object code which uses floating point instructions for both integer and real arithmetic. Hence, real numbers R and integer numbers N must lie in the following range.

$$10^{-38} \leq |R| \leq 10^{38} \quad \text{or} \quad R = 0$$

$$0 \leq |N| < 2^{27}$$

The internal representation of true is 777777777777 octal.
The internal representation of false is 000000000000 octal.

2) Large programs/arrays are limited by the core size of 32768 storage cells.

3) Extremely large programs may not be translated because of table overflow during compilation.

4) Due to a limited character set for the 7090/7094, many of the Reference ALGOL symbols are not available. Table 3.4-1 gives for each Reference ALGOL symbol, its hardware ALGOL symbol. Some of the hardware ALGOL symbols have alternate or ''tolerated'' symbols which may be used if desired. The hardware ALGOL symbols and ''tolerated'' symbols must be used when writing ALGOL programs to be compiled by the ALGOL translator which operates under MAMOS.

Table 3.4-1: REFERENCE TO HARDWARE REPRESENTATION

ALGOL Symbol	Symbol Name	Hardware Symbol	Tolerated
A|B|...|Z	upper case alphabet	A|B|...|Z	
a|b|...|z	lower case alphabet	A|B|...|Z	
0|1|...|9	numerals	0|1|...|9	
+	plus sign	+	
-	minus sign	-	
×	multiplication sign	*	
/	division sign	/	
÷	integer division sign	//	
:	colon	..	
;	semi-colon	.,	
(	left parenthesis	(	
)	right parenthesis	)	
[	left bracket	(/	
]	right bracket	/)	
,	comma	,	
.	decimal point	.	
:=	assignment sign	.=	=
↑	exponentiation	'POWER'	**
<	less than	'LESS'	'LS'
≤	less than or equal to	'NOT GREATER'	'LQ'
=	equal to	'EQUAL'	'EQ'
≥	greater than or equal to	'NOT LESS'	'GQ'
>	greater than	'GREATER'	'GR'
≠	not equal to	'NOT EQUAL'	'NQ'
≡	logical equivalent	'EQUIV'	'EQV'
⊃	logical implies	'IMPL'	'INP'

Table 3.4-1: REFERENCE TO HARDWARE REPRESENTATION - Continued

ALGOL Symbol	Symbol Name	Hardware Symbol	Tolerated
∨	logical or	'OR'	
∧	logical and	'AND'	
¬	logical negation	'NOT'	
10	base 10	'(apostrophe)	
#|b	blank space		
(	left string quote	'('	''
)	right string quote	')'	''
true	Boolean true	'TRUE'	
false	Boolean false	'FALSE'	
go to		'GO TO'	
if		'IF'	
then		'THEN'	
else		'ELSE'	
for		'FOR'	
do		'DO'	
step		'STEP'	
until		'UNTIL'	
while		'WHILE'	
comment		'COMMENT'	
begin		'BEGIN'	
end		'END'	
boolean		'BOOLEAN'	
integer		'INTEGER'	
real		'REAL'	
array		'ARRAY'	
switch		'SWITCH'	

Table 3.4-1: REFERENCE TO HARDWARE REPRESENTATION - Continued

ALGOL Symbol	Hardware Symbol
procedure	'PROCEDURE'
string	'STRING'
label	'LABEL'
value	'VALUE'
code	'CODE'
finis	'FINIS'

Note that the ALGOL word symbol own is not in the above table because own has not been implemented in the current version of ALGOL. Also note that two ALGOL word symbols, code and finis have been added. The symbol 'FINIS' is used to denote the end of a program to be compiled by the ALGOL translator. 'FINIS' must be used.

The symbol 'CODE' is used to replace the procedure body of a procedure which is compiled independently by ALGOL, or some other translator.

Definition of a Source Program to be Compiled by ALGOL under MAMOS

An ALGOL source program is a program as defined in the ALGOL report (section 4.1.1.) followed by the word symbol 'FINIS', i.e.

<Algol source program> ::= <program> finis

An ALGOL source program will fall in one of the following categories:

1) The source program may be complete, in which case it will contain any necessary procedures. In other words, if the program is correct; and it is compiled and executed, then the desired result will be produced.

2) The source program may be complete except for one or more procedure bodies. This type of source program is produced when 'CODE' is substituted for a procedure body.

3) The source program may be a procedure.

Punching the ALGOL Source Program

An ALGOL source program is punched in card columns 1-72 of as many cards as desired. Blanks are ignored (except for H fields in FORMAT procedures). Card columns 73-80 are ignored by the ALGOL translator and these columns may be used for identification or any other purpose, as long as any punching in 73-80 consists of legal 7090/7094 BCD characters. Since ALGOL statements are separated by the semi-colon or word symbols, several statements (or only part of a statement) may be punched on a single card. The last symbol punched is 'FINIS'.

The deck of cards which results from punching an ALGOL source program will be called an ALGOL source deck.

MAMOS Control Cards

MAMOS control cards are identified by having a dollar ($) sign punched in column 1. Control specifications are punched on the control cards in columns 2 through 64. If more than one specification is punched on a single control card, they are separated by commas. The specifications indicate to MAMOS what is to be done as the job is processed.

Necessary Control Specifications

$ COMPILE ALGOL

Every ALGOL source deck must be immediately preceded by the above specification which indicates to MAMOS that the ALGOL compiler is needed.

$ EXECUTE

The above specification must be used if execution of a program is desired.

$ DATA

If execution is desired and execution requires a data deck [DATA], then the data deck must be preceded by the $DATA specification.

Optional Control Specifications

$ PUNCH OBJECT

The above specification is used if it is desired to get as part of the output a binary deck of the object programs generated by the ALGOL compiler. The specification must be given for every ALGOL source deck for which a binary deck is desired.

Note on EXECUTE and DATA specifications

The $EXECUTE and $DATA specifications may give trouble if MAMOS is operating under DC-IBSYS, and either specification is punched starting in Column 2. A convention of leaving Column 2 blank for the above specifications is desirable for DC-IBSYS users.

Composition of Simple ALGOL Job Decks

The most simple application of ALGOL under MAMOS would consist of a compilation of one ALGOL source program. The job deck would be made up as follows:

```
$EXECUTE        MAMOS
$ID   NAME*TASK*OPTIONS $ COMMENT
$ COMPILE ALGOL
   [Algol source deck]
```

The output from the above job would consist of a listing of the source program and any diagnostics detected by the ALGOL compiler.

If we replaced the $ COMPILE ALGOL card above with the card

```
$ EXECUTE,COMPILE ALGOL
```

and if the ALGOL source program compiled correctly, then execution of the compiled program would be attempted. Output from the job would be the same as above, plus any output generated during execution. Note that 2 or more specifications may appear on the same card as long as they don't go beyond card column 64. It is assumed above that no data is required for execution of the program.

Example

The following example of an ALGOL job deck uses all of the 4 specifications described above. The program reads the number X, computes y=F(X) = $\sqrt{X}$, then prints X and F(X). These 3 operations continue until the data deck is all read in and then the job is automatically terminated by MAMOS.

Computation of F(X) is done by means of a procedure.

```
$EXECUTE        MAMOS
$ID   DOSSIE*XXX/YY/ZZZ*5S*1P$COMMENT
$COMPILE ALGOL, EXECUTE, PUNCH OBJECT
  'BEGIN' 'COMMENT' SIMPLE EXAMPLE.,
     'REAL' X,Y.,

    'PROCEDURE'  FOFX(X,Y)., 'REAL' X,Y.,
         'BEGIN' Y.=SQRT(X)., 'END' FOFX.,

  L..READ(X).,
     FOFX(X,Y)., PRINT(X,Y).,
    'GO TO' L.,
     'END'
     'FINIS'
$DATA
 1.0,10.0   0.31415927'1        10.000001'-1
```

The READ and PRINT procedures are described later.

In the above example, the first two cards are IBSYS control cards which are necessary for any job run under MAMOS.

The $ID card above specifies that execution of the program is not to be allowed to run more than 5 seconds, and that no more than 1 page of output will be generated during execution. See section 1.2 for a complete description of the $EXECUTE and $ID IBSYS control cards.

The third card specifies that (1) an ALGOL program which follows is to be compiled, (2) execution of the program is desired, and (3) the binary cards produced by the ALGOL compilation are to be given as part of the job output.

The fourth card and all cards up to but not including the $DATA card consist of the ALGOL source program which is to be compiled.

The $DATA card specifies that execution, if no errors have been detected, is to begin.

The last card is a data card which has 4 values of X punched in a free form.

An outline is given below of the output produced when the above job deck is run.

Printed output

1) The $ID and sign-on information.
2) The MAMOS specifications.
3) A list of the ALGOL source program.
4) The $DATA card is printed.
5) If no errors were detected during compilation then execution output consisting of 4 lines of X, F(X). Also, just previous to the execution results is a printed list of subroutines used and their octal origins in core.
6) The $ID and sign-off information.

Punched output

1) The $ID card and a binary deck of the instructions compiled for the ALGOL source program. The binary deck is not given if an error is detected during compilation of the ALGOL source program.

The 'CODE' Procedure

Procedures may be compiled independently of the ALGOL source program which calls them.

An ALGOL program may call an independently compiled procedure by means of a 'CODE' procedure. The 'CODE' procedure is a regular procedure with the body of the procedure replaced by the ALGOL symbol 'CODE'.

Thus the above example of a job deck could be written as follows to produce the same results.

```
$EXECUTE        MAMOS
$ID    DOSSIE*XXX/YY/ZZZ*5SECONDS*1PAGE$
$COMPILE ALGOL, EXECUTE, PUNCH OBJECT
   'BEGIN''COMMENT'SIMPLE EXAMPLE.,
       'REAL'X,Y.,
       'PROCEDURE'FOFX(X,Y).,'REAL'X,Y.,'CODE'.,
    L.. READ (X)., FOFX(X,Y)., PRINT(X,Y).,
       'GO TO' L., 'END' 'FINIS'
$COMPILE ALGOL, PUNCH OBJECT
   'PROCEDURE' FOFX(X,Y)., 'REAL'X,Y.,
      'BEGIN'Y.=SQRT(X).,'END'FOFX.,
   'FINIS'
$DATA
 1.0,10.0    0.31415927'1         10.000001'-1
```

Restriction On 'CODE' Procedures

'CODE' procedure names must be less than 7 characters in length.

INPUT/OUTPUT

The following description of input/output for ALGOL programs is taken almost exclusively from the Illinois Users Manual mentioned above.

INPUT/OUTPUT IN ALGOL PROGRAMS

There is no specification in the ALGOL Report for input/output operations in ALGOL. This was not an oversight on the part of the designing committee, but a result of its realization that input/output operations vary so much from one installation to another and from one computer to another that specifications for input/output were better left to each installation. Hence, the ALGOL Translator uses code procedures for input/output. The use of these procedures is described in detail below.

Code Procedures for Input/Output.

There are several ALGOL code procedures which are associated with the input/output operations presently available through the ALGOL Translator. These basic I/O procedures are viewed in the same light as standard functions; that is, they are considered to have such importance and universal applicability that they are global to all ALGOL programs compiled by the Translator. For the user this means that there is no need to declare the input/output procedures. It further implies that the identifiers used for these procedures must have the same restricted use as those set aside for the standard functions, sin, cos, exp, etc. To use the identifiers for any other purpose can cause an error condition. However, one can ''submerge'' any of these procedure names by declaring a procedure or variable with the same name, as one can do with ordinary identifiers in nested blocks.

For example,

```
begin real a, b, c;
      read (a, b);
      c:= a + b;
      print (a, b, c)
end
```

shows the use of the read and print code procedures. Neither has been declared in the example, since this is unnecessary.

On the other hand,

```
begin real a, b, c, d;
      read (b); begin
                      procedure read (e,f);
                            real e, f.,
                         e:= f↑2; read (a, b)
                end;
      read (d); c:= a + b;
      print (a, b, c, d)
end
```

shows an entirely different use of a declared procedure with the same name as read.

This procedure is declared in an inner block, used there, and is no longer defined after exit from that block. Hence the statement ''read (b)'' causes the real number b to be read; ''read (a,b)'' causes the calculation a:= b↑ 2 to be made; and ''read (d)'' causes the real number d to be read.

Simplified Input/Output

Since ALGOL is a language designed for expressing algorithms in numerical analysis, input and output operations are concerned mainly with the transmission of numerical data.

There are two input procedures and two output procedures designed especially for the user who does not have specific format requirements.

The two simplified input procedures are read and readmatrix, and both accept data in a free form. The form of the read procedure call is

```
read (a, b, c,...)
```

where a, b, c,... are variables, either simple or subscripted. The procedure reads one variable at a time, so if subscripted variables appear in the list, then subscripts must be specified. For example, let a be an array of dimension 3 x 4 and b and c be simple variables. Then

```
read (a, b, c)
```

is incorrect, while

```
read (a[1,2], b, c)
```

is acceptable. Of course, in the last case, only element a [1,2] will be read, and not the entire array.

If the user has an entire array to be read, a second easy-to-use procedure is available, readmatrix. The form of its call is

```
readmatrix (a, b, c,...)
```

where a, b, c,... are array identifiers. The procedure reads elements of an array in such a way that the last index changes first, then the preceding one, etc.

The input data in both cases is assumed to be in a free form. The data can be any ALGOL number (see the ALGOL Report) and placed anywhere on a card. The numbers are separated by three blanks, a comma, or the end of the card (column 72). Successive calls for either of the procedures does not initiate reading from a new card; reading proceeds continuously from one number to the next on a card and when that card is exhausted (column 72) it proceeds to the next.

The two output procedures for simplified use are print and printmatrix. The form of the print call is

print (E_1, E_2, ...)

where E_1, E_2, ... represent arithmetic expressions. Of course, an arithmetic expression may consist of simply a variable name, and in most cases it probably will, so

print (area, depth, velocity * weight)

is an acceptable print procedure call. All the output from such a call will be printed on the off-line printer according to the standard format list

'1X,5E14.7'

That is, 5 numbers per line will be printed, each with 7 digits to the right of the decimal point, in what is commonly called ''scientific notation''. The number -3765.831 would appear in this notation as

-.3765831E 04

and the number .00376 becomes

.3760000E-02

The printmatrix procedure call is

printmatrix (a, b, c,...)

where a, b, c,... are names of arrays. Output is by rows in exactly the same format as that of print, 5 elements per line. The 3 x 4 array b would be printed as

b_{11} b_{12} b_{13} b_{14} b_{21}

b_{22} b_{23} b_{24} b_{31} b_{32}

b_{33} b_{34}

No alphabetic data can be input or output with any of the four simplified procedures.

For more control over the format of the input and output, other procedures are available and are described in the following sections.

At this point, it appears desirable to begin using certain unfamiliar terms and notation, such as ''syntax'' and ''semantics'' and unconventional brackets < > and vertical lines |. These conventions have been borrowed from the field of linguistics and are highly useful in describing precisely how parts of a language (and ALGOL is a language, however limited it may be) can be put together to mean something to someone or something. The reason for including these conventions here is mainly to be precise in describing certain things omitted by the ALGOL Report, but also to initiate the ALGOL beginner in the terminology of the ALGOL Report. Ability to read and understand the Report will be indispensable to the active ALGOL user, so an attempt to entirely avoid the notation problem would be false economy. If the reader keeps in mind these interpretations of the symbols, he should progress well.

::= means ''is''

| means ''or''

< > are simply brackets that mean that the terms enclosed by them go together to form a single unit.

For example,

<unsigned integer> ::= <digit> | <unsigned integer> <digit>

can be read ''An unsigned integer is either a digit or an entity composed of an unsigned integer followed by a digit''. This is simple enough, but the definition is strange in that it uses ''unsigned integer'' to define ''unsigned integer''. This is a recursive definition and is quite simple to explain an unsigned integer is either a single digit (0,1,2,...,9) or an entity composed of a digit following one or more digits. With these conventions in mind, we proceed to an exposition of the more comprehensive input and output procedures.

The Format Procedure

The format procedure provides the basic information to the input/output procedures for the placement and scaling of information, whether it is on a card image as input or on a printed page as output.

In the following, the complete syntax of the format procedure is given in the same notation used for the ALGOL Report; a discussion of the meanings and uses of the various constructions completes the coverage of the format procedure.

Syntax.

<format call> ::= FORMAT (<integer expression>, <format list>)

<format list> ::= <format string> | <format list>, <format string>

<format string> ::= <left string quote> <secondary list> <right string quote>

<secondary list> ::= <secondary> | <secondary list>, <secondary>

<secondary> ::= <field specifier> | (<format primary>) | <unsigned integer> (<format primary>)

<format primary> ::= <field specifier> | <format primary>, <field specifier>

<field specifier> ::= <F-conversion> | <E-conversion> | <X-field> | <H-field> | <void-specification> | <record separator>

Semantics.

<format call>: The form of the format procedure call is

FORMAT (E, '('A, B, C, ...')') ,

where the E represents an integer expression and the list of indefinite length, A, B, C, ..., represents units of information concerning the form of data. The integer expression denoted by E above identifies a logical tape unit available to the user. It is the responsibility of the user to satisfy this requirement.

The tape numbers designated by the integer expression E correspond to the MAMOS logical tape units as follows:

E	MAMOS Logical Unit	Use
1	7 (input) or 6 (output)	regular input (output) tape
2	2	scratch tape
3	3	scratch tape
4	4	scratch tape
5	5	regular punch tape
6	6	regular print tape
7	7	regular input tape
9	9	scratch tape
10	10	special input/output tape
11	11	special input/output tape

The term ''scratch tape'' in the table means that during execution those tapes are available to the user for whatever use he wishes.

The number E=1 is a special all-purpose parameter which, when used, automatically causes designation of the regular input tape (logical 7) if the call is readf or readmatrixf, or the regular print tape (logical 6) if the call is printf or printmatrixf.

<format list>: This is a list of ALGOL strings separated by commas. No fixed number of such strings is required in a format call, in contrast to the normal procedure call. That is, the format procedure is considered to have an arbitrary number of formal parameters.

Each of the strings must be enclosed in string quotes, and might appear as '('A, B, C,...')', where A, B, C, ... represents a list (of arbitrary length) of units of information concerning the form of data. These units of information are field-specifiers, which prescribe a form for data, or collections of field-specifiers enclosed in parentheses. The field-specifiers provide for input or output of (1) numerical data in the familiar decimal notation (as 123.76) or in ''scientific notation'' or exponential form (as $.12376 \times 10^3$), (2) blank fields, and (3) alphabetic-numeric information, such as titles, headings, notes to the user, etc., or act as record separators.

<secondary>: The secondary exists for two important reasons. Both are concerned with the use of a portion of a format list more than once for a given input or output procedure call. To be realistic here, we must assume that the secondary consists of several field-specifiers enclosed by parentheses, and perhaps preceded by an unsigned integer. Such a secondary might appear as

$$3(P_1, P_2, P_3)$$

where the P_i are field specifiers. This has the same effect as the format list

$$(P_1, P_2, P_3), (P_1, P_2, P_3), (P_1, P_2, P_3)$$

and, except in the case mentioned below, the same effect as

$$P_1, P_2, P_3, P_1, P_2, P_3, P_1, P_2, P_3$$

The other use for the secondary enclosed by parentheses occurs when an input or output procedure call lists more variables than are listed in the controlling format procedure call. When the format list has been exhausted but the input or output list has not, then control of format goes back to the last left parenthesis before the end of the format list, and input or output proceeds according to the field specifiers to the right of this left parenthesis.

<primary>: The primary consists of a single field specifier or several field specifiers separated by commas. It should be apparent that in many cases a primary is also a secondary (e.g., when it consists of a single field specifier).

<record separator>: The record separator is a slash or a series of slashes. Since input is in the form of card images on magnetic tape, each slash in the format list causes reading of a new card image; for output, each slash causes a new line of printing or punching to be started. The first field of the new record is that specified by the first field specifier following the record separator. In general, n successive slashes will cause n - 1 blank lines on the printed output, or n - 1 successive cards to be ignored.

The format procedure call must account for every column in the unit record with which it is concerned. With input, the originating medium is a card, so every column on the card must be accounted for, beginning with column 1 and continuing through the last column containing information of interest. The Translator assumes that unaccounted for columns remaining to the right in a card image are of no interest. For example,

```
FORMAT (7, '('F 10.4, 3 F 15.6, 5 X, F 10.4, 10 X')')
```

accounts for all 80 columns on the card, even though the last 10 (71-80) columns are to be skipped and not read. We can accomplish exactly the same thing by

```
FORMAT (7, '('F 10.4, 3 F 15.6, 5 X, F 10.4')')
```

On the other hand, we cannot ignore leading blank fields (or X-fields, generally). Thus,

```
FORMAT (7, '('F 10.4, 5 X, F 10.4')')
```

and

```
FORMAT (7, '('10 X, F 10.4, 5 X, F 10.4')')
```

are not equivalent.

The same general idea is true for output, the essential difference being the fact that instead of reading card images, we are printing lines of characters, 132 characters per line, or punching cards, 80 columns per card, and every space must be accounted for. Again all unspecified spaces to the right of specified fields are left blank. The first character of every output record written on the regular output tape (logical tape 6) is used as a carriage control character, and is not printed. The carriage control characters are as follows:

Character	Meaning
blank	single space
1	skip to next page
2	skip to next half page
4	skip to next quarter page
6	skip to next sixth page
8	skip to next sixth page
0	double space
-	triple space
+	space suppress
9	suppress automatic page overflow.

Field Specifiers.

Syntax.

<F-conversion> ::= F <unsigned integer> . <digit> | <unsigned integer>
F <unsigned integer> . <digit>

<E-conversion> ::= E <unsigned integer> . <digit> | <unsigned integer>
E <unsigned integer> . <digit>

<X-field> ::= <unsigned integer> X

<H-field> ::= <unsigned integer> H <proper string>

<record separator> ::= /| <record separator>/

Semantics.

<F-conversion>.

The F-conversion field specifier is of the form nFw.d, where n , w , and d are unsigned integers. If n = 1, it may be omitted.

The n in this field specifier denotes the number of such consecutive fields; hence 3F10.3 is equivalent to

F10.3,F10.3,F10.3,

and 1F10.3 is equivalent to simply F10.3.

The w in this field specifier indicates the total width of the field in number of characters. The appearance of numbers in the F-conversion is the familiar form of a sequence of decimal digits in which there appears one, and only one, decimal point. Hence, the total characters in the field must include the decimal point. A number in this conversion may be either plus or minus, so w must also include one column count for the sign.

For input the plus sign may or may not be punched at the discretion of the user; the minus sign must be punched and must precede the most significant digit in the field.

For output, the plus sign will not be printed; the minus sign will be printed in the first column to the left of the most significant digit in the field. Leading zeroes will not be printed.

The d in the field specifier denotes the number of digits to the right of the decimal point. This number does not include space for the decimal point itself. d must not be greater than 20.

For example,

format (6, (F 8.4, F 6.2, F 10.3))

specifies a set of three fields, of 8, 6, and 10 columns respectively.

In the first, 4 digits lie to the right of the decimal point (which takes up one column itself). This leaves, of the original 8 columns, one more for the sign and 2 for digits to the left of the decimal point. In the second, 2 digits lie to the right of the decimal point and 2 to the left, leaving, of the original 6 columns, one for the sign and one for the decimal point. In the third, 3 digits lie to the right of the decimal point and 5 to the left.

Suppose we wish to print -12.1372, 21.63, and + 17238.312 according to the above format specification. With b representing blank spaces, our printed line would look like this:

```
|-12.1372|b21.63|b17238.312|
|        |      |          |
|field 1 |field2| field 3  |
```

<E-conversion>.

The E-conversion field specifier is of the form nEw.d, where n , w and d are unsigned integers. As with the F-conversion, if n = 1, it may be omitted.

The n in this field specifier denotes the number of such consecutive fields; hence 3 E 13.7 is equivalent to

E 13.7, E 13.7, E 13.7,

and 1 E 13.7 is equivalent to E 13.7.

Again paralleling the F-conversion, the w denotes the entire width of the field in number of characters. The appearance of numbers in the E-conversion resembles the form widely known as ''scientific notation,'' a decimal fraction followed by an exponent of 10, as, for example,

$$.78325 \times 10^3.$$

The exact form of numbers in the E-conversion is

$$\pm.dd...dE\pm ee,$$

where d's represent decimal digits, the E implies ''exponent follows'' and the ee represents a two digit exponent of 10. The two sign positions, one for the number itself and one for the exponent, are indicated by $\pm$. Note that every number in this conversion has at least six columns of its field used for ''bookkeeping'' symbols:

$$\pm\ .\ E\ \pm ee$$

Hence, if a field were specified as E13.7, the field would be 13 columns wide, only 7 of which can contain digits of the number put into this conversion. Similarly, E14.9 is an invalid field specification, since only 14 - 6 = 8 columns are available for digits of the number. A specification of E14.3 does not use all 8 of the columns available to it for placement of significant digits of the number.

For example, if we want to place -138,714.31 into E-conversion form in a field 14 columns wide, we specify E14.8, and we have

-.13871431E 06.

A field specification of E12.6 results in

-.138714E 06,

and one of E14.6 results in

bb-.138714E 06

In both these last cases, information has been lost in the conversion (the last two digits, 31, of the original number).

The F-conversion and E-conversion are the only conversions presently provided with ALGOL for input/output of numerical information, and integers as data have not been mentioned. There is no special integer conversion, but integers can be handled through either the F-conversion or the E-conversion. For example, the integer 317 becomes, in F5.0 conversion

bb317

It is important to note that the sign must be accounted for. The same number in E9.3 becomes

b.317E 03;

and in this case, we have had to provide for the 6 character spaces always present in the E-conversion, d must not be greater than 20.

<X-field>.

The X-field specifier is of the form nX, where n is an unsigned integer. The X-field is a field of n blank spaces. The n cannot be omitted, even if it equals 1.

The X-field makes it easy to space printed output as desired, and permits skipping of unwanted information on input cards. For example, suppose we have cards with six 10-column fields (beginning in column 1) and we wish to read only from the second, third and fifth fields. Assume the data in these fields are in F10.4 conversion. The format call will look like this:

```
FORMAT(6,'('10X, 2 F 10.4, 10X, F 10.4')')
```

A readf call of

```
readf (A, B, C)
```

will cause the data in fields 2, 3 and 5 to be stored as variables A, B and C, respectively. Note that in the format call above, the sixth field has not been accounted for, and need not be.

<H-field>.

The H-field specifier is of the form

nHss...s,

where the n is an integer and the ss...s is a proper string; i.e., the ss...s is a list consisting of any n characters available in the character set, except the escape symbols.

The use of the H-field is primarily to print labels, titles, variable names, etc., so as to make interpretation of printed output easier. For example,

```
FORMAT (7,'('23 HbCOMPUTEDbAVERAGESbb=bb, F 12.4')').,
PRINTF  (AVG)
```

will cause the 23 characters, including blanks, following H to be printed, followed by the current value of the variable AVG in F12.4 conversion. If AVG = 138.7642, we would have

bCOMPUTEDbAVERAGESbb=bbbbbb138.7642

as the printed output.

The user is responsible for assuring that n is precisely the number of characters he intends to be in the H-field.

The Input and Output Procedures.

The input and output procedures must each be preceded by a format procedure call in order for the computer to be able to correctly position and scale the input or output information, as the case may be. The set of simplified input/output procedures assumes a standard format, so that the user need not concern himself with providing formats for them. Indeed, he cannot, since the simplified procedures ignore all formats. Complete information on all input/output procedures follows.

Syntax.

<read call> ::= READ (<input list>)

<readf call> ::= READF (<input list>)

<readmatrix call> ::= READMATRIX (<array identifier list>)

<readmatrixf call> ::= READMATRIXF (<array identifier list>)

<input list> ::= <variable> | <input list>, <variable>

<array identifier list> ::= <array identifier> | <array identifier list>, <array identifier>

Semantics.

<read call>: The form of the read procedure call is

read (a, b, c, ...)

where a, b, c, ... represents a list of variables, simple or subscripted, separated by commas. They are read from card images on the input tape (logical 7) ignoring any format calls which may appear in the program. The procedure does not start reading automatically from a new card, but accepts ALGOL numbers in any defined form (see the ALGOL Report, section 2.5, Numbers), separated by a comma, three blanks, or the end of a card (column 72), continuously until the input list is exhausted. Further calls for the read procedure cause continuation of reading the same card, not for a new card.

<readf call>: The form of the readf procedure is identical to that of the read call. The difference between the two is that the readf procedure reads input according to the last executed format procedure call.

<readmatrix call>: The form of the readmatrix procedure call is

readmatrix (a, b, c, ...)

where a, b, c, ... are array identifiers. The procedure reads elements of an array in such a way that the last index changes first, then the preceding one, etc. The elements are acceptable in any ALGOL number form, separated by three blanks, a comma, or the end of a card (column 72).

<readmatrixf call>: The form of the readmatrixf procedure call is identical to that of the readmatrix. The difference between the two is that the readmatrixf procedure reads input according to the last executed format procedure call.

<input list>: The form of the input list is

A, B, C, ...

where A, B, C, ... represents a series of identifiers. They may be simple variables or elements of an array; in the latter case, the subscripts must be present, as for example A(/2,3/) and B(/7,6/). The array identifier above, without the subscripts, is not acceptable.

The user should keep in mind that the format procedure call not only controls the form of the data but also prescribes the logical tape number from which the data is read.

Syntax.

<print call> ::= PRINT (<output list>)

<printf call> ::= PRINTF (<output list>)

<printmatrix call> ::= PRINTMATRIX (<array identifier>)

<printmatrixf call> ::= PRINTMATRIXF (<array identifier>)

<output list> ::= <arithmetic expression> | <output list>, <arithmetic expression>

Semantics.

<print call>: The form of the print procedure call is

print (E_1, E_2, ...)

where E_1, E_2, ... represent arithmetic expressions. The procedure evaluates the arithmetic expressions at execution time and places the results on the output tape for the off-line printer according to the standard format list

'1X, 5E14.7'

<printf call>: The form of the printf procedure call is

printf (E_1, E_2, ...)

where the E_1, E_2, ... represent arithmetic expressions. Despite its name, the procedure can be used for various output tasks, such as placing intermediate results on scratch (utility) tapes, placing card images on the punch output tape for punching into cards, or printing output on the off-line printer, depending upon the logical tape unit prescribed by the last executed format procedure call preceding the printf procedure call which also controls the data transmitted.

<printmatrix>: The form of the printmatrix procedure call is

printmatrix (a, b, c, ...)

where a, b, c, ... are array identifiers. The elements of the array are printed on the off-line printer according to the standard format list

'1X,5E14.7'

Hence, the 2 x 3 matrix a will be printed as

a_{11} a_{12} a_{13} a_{21} a_{22}

a_{23}

<printmatrixf>: The form of the printmatrixf procedure call is

printmatrixf (a, b, c, ...)

where a, b, c, ... are array identifiers. The elements of the array are output to the tape unit specified by the last executed format procedure call preceding the printmatrixf procedure call, which also controls the format of the data thus transmitted.

<output list>: The output list consists of arithmetic expressions of any kind, separated by commas, but cannot be void. That is, an output procedure such as

printf ( )

is not valid, even though the controlling format may consist entirely of an H-field.

3.5 MAMOS job deck composition in general

This section describes the various compositions of legal MAMOS job decks. The IBSYS control card $ID is not specified in full for the illustrations and the user should refer to section 1.2 for a description of the $ID card.

The MAMOS specifications have been defined in section 3.2, but some will be elaborated on in later sections.

DEFINITIONS

[SOURCE] will be used to indicate a deck which consists of at least one MAMOS specification followed by a source program which requires a translation by some translator of the MAMOS system.

[OBJECT] will be used to indicate a relocatable binary deck of a type produced by the MAD compiler, the FORTRAN II compiler, the ALGOL compiler, or the UMAP assembler.

[DATA] will be used to indicate a data deck to be used by the program during execution.

[CORELOAD] will be used to indicate a deck which consists of one or more [SOURCE]s and/or [OBJECT]s. The last card of a [CORELOAD] is the specification card
$BREAK
or
$BREAK(N)
[CORELOAD] is used in PING PONG jobs, and must contain one and only one main program.

[BINARY] will be used to indicate a deck composed of the specification card
$BINARY
followed by one or more [OBJECT]s.

[X]i will be used to indicate the ith [X] in the job deck. X is SOURCE, OBJECT, DATA, BINARY, or CORELOAD.

EXAMPLES OF A [SOURCE]

(1) The following deck is an example of a [SOURCE] which requires a MAD compilation.

```
$ COMPILE MAD
          EXECUTE SYSTEM.  ] Could be any legal MAD
          END OF PROGRAM   ] source deck.
```

The MAD program above does nothing useful but could be of any length and complexity desired. Also, any or all of the specifications below could be used. The allowable specifications are PRINT OBJECT, PUNCH OBJECT, CONDITIONAL, and also binary card labeling information may be given in columns 65-72 of a specification card.

(2) The following deck is an example of a [SOURCE] which requires a UMAP assembly.

```
$ ASSEMBLE
*  UMAP  PROGRAM WHICH DOES NOTHING.  ] Could be any legal
       CALL    SYSTEM                 ] UMAP source deck.
       END
```

Other specifications which may be used in the UMAP [SOURCE] are PUNCH OBJECT and CONDITIONAL. Also, binary card labeling information may be given in columns 65-72 of a specification.

(3) The following deck is an example of a [SOURCE] which requires a FORTRAN II to MAD translation by the MADTRAN translator, followed by a MAD compilation of the MAD source produced by MADTRAN.

```
$ MADTRAN
C FORTRAN PROGRAM TO DO NOTHING.  ] Could be almost any
       CALL EXIT                  | legal FORTRAN II source
       END                        ] deck.
```

Other specifications which may be used are the same as those options allowed in example (1) above.

PING PONG

PING PONG is the process of optionally calling and passing control to one of several complete programs ([CORELOAD]s) which comprise a PING PONG job. A PING PONG job usually is a job whose program is too large to fit in core memory at one time. If a large program can be broken into segments or [CORELOAD]s which are independent of each other except for communication through a common area of memory or external storage, then the program may be run as a PING PONG job.

Each [CORELOAD] must logically end by either calling in another [CORELOAD] or terminating the job. A library subroutine, with two entries, is available for the purpose of calling in a new coreload.

Before execution begins, the program for each [CORELOAD] is stacked on logical tape T where T is specified on the last card of a [CORELOAD], or if not specified T is assumed to be logical tape 2. If T is specified, it must be 2, 4, or 9. The order of stacking on the tapes is in the order the [CORELOAD]s appear in the job deck.

To call the next [CORELOAD] in sequence on logical tape T, the following call is given.

```
    SEQPGM.(T)        for MAD or
CALL    SEQPGM,T      for UMAP.
```

To call [CORELOAD] R on logical tape T, the following call is given.

```
    SELPGM.(R,T)      for MAD or
CALL    SELPGM,R,T    for UMAP.
```

If T is omitted in the above calls then logical tape 2 will be used.

At the start of execution a specific [CORELOAD] must be automatically chosen by MAMOS as the first [CORELOAD] to load and give control to. The [CORELOAD] which is normally chosen as the first to execute is the first one which physically appears in the job deck. However, if the $BREAK specification is omitted from the last [CORELOAD] in the job deck, then that [CORELOAD] is executed first, without being stacked on a tape. Hence, in the later case the last [CORELOAD] could only be executed once. An example of a PING PONG job is given later.

JOB DECK

The composition of a general job deck to be run under MAMOS is now given.

Non-PING PONG job deck

```
$EXECUTE        MAMOS
$ID
$options
.
.
.
$options
   [SOURCE]1
      .
      .
      .
   [SOURCE]L
   [OBJECT]1
      .
      .
      .
   [OBJECT]M
$BINARY
   [OBJECT]M+1
      .
      .
      .
   [OBJECT]M+N
$DATA
   [DATA]
```

PING PONG job deck

```
$EXECUTE        MAMOS
$ID
$options
.
.
.
$options
   [CORELOAD]1
      .
      .
      .
   [CORELOAD]K
$DATA
   [DATA]
```

The options above may be on as many $ cards as desired. The allowable specifications at this point in the job deck are EXECUTE, COPIES, DUMP, FULL DUMP, I/O DUMP, SUB TRACE, HALT, and comments which appear on cards with $$ in columns 1 and 2.

The TEACHER, UNKNOWN, and REGRESSION specifications could also be included in the above list, but since these three specify special jobs they are described later.

TEACHER JOB

A TEACHER job is composed as follows.

```
$EXECUTE        MAMOS
$ID
$TEACHER
$$ comments if desired
   [OBJECT]1
      .
      .
      .
   [OBJECT]n
$DATA
   [DATA]
```

Execution of the TEACHER job causes a pseudo-input tape to be written on logical tape 10. The tape will consist of the object programs and data supplied with the TEACHER job.

The above deck may consist of one main program, any number of sub-programs, and a data deck if desired. Any main or sub-program must be in binary form. A teacher job which supplies only $DATA and [DATA] is also permissible.

UNKNOWN JOB

This job is the type of job which follows a TEACHER job, and is connected to and uses the program(s) and or data which was put on logical tape 10 by the TEACHER job.

An UNKNOWN job is composed as follows.

```
$EXECUTE        MAMOS
$ID
$options
.
.
.
$options
$UNKNOWN
   [SOURCE]1
      .
      .
      .
   [SOURCE]L
   [OBJECT]1
      .
      .
      .
   [OBJECT]M
```

This job is the same as the general job defined previously except--

1) There is no $DATA specification or [DATA].
2) The specification UNKNOWN is used.

MADTRAN Under MAMOS

MADTRAN is a program written in MAD language which translates a FORTRAN II source program into a MAD source program.

The $ MADTRAN specification causes the following FORTRAN II source program to be translated to a MAD source program. The MAD source is then compiled by the MAD translator. In other words,

```
$ MADTRAN
  [FORTRAN II source]
```

is equivalent to

```
$ COMPILE MAD
  [MAD source]
```

when the FORTRAN II, and MAD source decks do the same task. Of course the object programs may not be physically the same.

The output of the MADTRAN translator consists of a listing of the FORTRAN II source program, a table of corresponding MAD and FORTRAN statement labels, a list of the function names used in the MAD program, and a symbolic deck of the equivalent MAD program.

MAD statement labels of the form QQnnnn are created during the translation. The user can specify a different alphabetic 2 character label of the form XXnnnn by use of the statement

```
STATEMENT LABEL IS XX
```

This statement, if used, must precede the first statement of the FORTRAN II source program, and XX must be two alphabetic characters which never appear in the FORTRAN program as a variable name of the form xxnnnn.

The output MAD source deck will normally have identification labels in card columns 73-80 of the form MTRnnnnn,nnnnn=00001,00002,.... The user can specify a different card label by use of the statement

```
CARD ID IS xxxnnnnn
```

in which case the card label in 73-80 will be xxxnnnnn, xxxnnnnn+1,.... nnnnn must be numeric. This statement, if used, must precede the first statement of the FORTRAN program.

The output from the MAD compilation which automatically follows a MADTRAN translation depends upon the specifications which are used. PRINT OBJECT and PUNCH OBJECT have the same effect as when given for a MAD compilation. The EXECUTE specification can also be used if desired.

Restrictions on MADTRAN

1) The FORTRAN II statements

SENSE LIGHT i
IF (SENSE LIGHT i) n_1,n_2
IF (SENSE SWITCH i)n_1,n_2
IF ACCUMULATOR OVERFLOW n_1,n_2
IF QUOTIENT OVERFLOW n_1,n_2
IF DIVIDE CHECK n_1,n_2

should be avoided if possible. MADTRAN will produce calls to a special subroutine to accomplish the results of such statements, and the user of these statements will have to supply the subroutine.

2) MADTRAN makes an assumption in its treatment of argument lists. If an argument is the name of an array and appears unsubscripted in the list, the zero element in the equivalent MAD array, hence the entire array, will be the argument and it may be subscripted in the MAD external function. If, however, the argument is a subscripted array name, the effective argument in MAD will be the value of that element.

3) FORTRAN COMMON is assigned to the MAD PROGRAM COMMON. Equivalence is handled as in MAD, so there is no reordering implied by the EQUIVALENCE statement.

4) Arrays in MAD are stored by rows rather than by columns as in FORTRAN II.

5) MADTRAN does not handle double precision (D in column 1), Complex (I in column 1), and Boolean (B in column 1) FORTRAN II statements.

6) The FORTRAN II source program must have

a) no more than 2000 statement labels.
b) no more than 1000 integer variables.
c) no more than 300 dimensioned variables.
d) no more than 1000 function names.
e) no more than 20 arguments in a one-line function definition.

7) FORTRAN II programs which use absolute logical tape numbers will have to be changed to refer to the MAMOS logical numbers as follows.

SYSIN1 = logical tape 7 for regular input tape.
SYSOU1 = logical tape 6 for regular output tape.
SYSPP1 = logical tape 5 for regular punch tape.

Logical tapes 2, 3, 4, 9, 10, 11 are available for scratch or special input/output.

8) The FORTRAN II PRINT and PUNCH statements do not at present compile properly if there is no list in the statements.

REGRESSION JOB

A REGRESSION JOB is composed as follows:

```
$EXECUTE        MAMOS
$ID
$ REGRESSION, EXECUTE
  [Transformation if desired]
$ DATA
  Control Card
  Format Card(s)
  Data Set(s)
  Blank Data Set
```

The Regression program is written as a FORTRAN II subroutine and it is called from the MAMOS library automatically. A description of the program follows.

REGRESSION ANALYSIS PROGRAM

The objective of the regression analysis program is to generate a linear regression equation of the form

$$(1) \quad Y=B_0+B_1X_1+B_2X_2+\ldots+B_KX_K$$

from a set of N observations of a set of K independent variables $(X_1,X_2,\ldots X_K)$ and a single dependent variable Y.

The coefficients $B_0,B_1,\ldots,B_K$ are determined so that the regression equation minimizes the sum of the squares of the deviations between the observed and predicted values of Y. The method used is the stepwise regression procedure which generates the expression (1), variable by variable, in order of relative importance, until all significant variables are included in the equation, according to the level of significance specified by the user.

This writeup includes both a simple linear regression procedure and a procedure to utilize transformations of the variables (as desired by the user) in forming the regression equation. Both programs have been combined into a single program. The use of this program is described in two parts, but the user must remember that the same program is used in each case, under the control of the parameters supplied.

1. LINEAR STEPWISE REGRESSION PROGRAM

The most common use of the regression procedure is the generation of a linear regression equation of the form (1). In this use, the variables are treated without applying any transformation functions before forming the regression equation.

CONTROL CARD

The Control Card is the first card of each problem presented to the computer. By the use of the variables punched on the control card the user may control the operation of the program on the data sets which follow. The card is described below, field by field, giving the name of the variable whose value is to be entered, the columns to be used and the type of variable (i.e., integer (I) which must be punched without using the decimal point characters or floating point (F) which must have the decimal point character punched). Following this, a brief description of the function of each variable is given.

VARIABLE NAME	COLUMNS	TYPE	FUNCTIONAL DESCRIPTION OF VARIABLES
TOL	1 - 10	F	Tolerance (normally .001), used as a bound on divisors used in matrix transformations. No variable having diagonal matrix element less than this value, at any stage of analysis, may be entered in the regression equation.
FLVLIN	11 - 20	F	F level for entering variable (see discussion of F level and significance).
FOUT	21 - 30	F	F level for removing variable (must be less than or equal to FLVLIN).
PROBNO	31 - 35	I	Problem number.
NOVAR	36 - 40	I	Total number of variables (including dependent variable), must be less than or equal to 101.
NODATA	41 - 45	I	Number of data sets (N). (No practical upper limit).
IFRTN	47	I	Blank or 0 causes the regression analysis to attempt to process successive regression problems from the input tape. (This is the normal case), 1 causes a return to the calling program at the completion of the current problem. This is effective on every new control card. (See the REGRESSION ANALYSIS PROGRAM AS A SUBROUTINE).
IFWT	48	I	Blank or 0 if weight factors given, 1 if data sets are all of unit weight.
IFSTEP	50	I	If data sets are all of unit weight. Blank or 0 causes printing of each step of regression analysis, 1 suppresses this printing (Final step is always printed).

VARIABLE NAME	COLUMNS	TYPE	FUNCTIONAL DESCRIPTION OF VARIABLES
IFRAW	52	I	Blank or 0 causes printing of raw sums, sums of squares and sums of cross products, 1 suppresses this printing.
IFMEAN	54	I	Blank or 0 causes printing of means and standard deviations, 1 suppresses this printing.
IFRESID	56	I	Blank or 0 causes printing of residual sums of squares and cross products adjusted about the means, 1 suppresses this printing.
IFCOR	58	I	Blank or 0 causes printing of simple correlation matrix, 1 suppresses this printing.
IFPRED	60	I	Blank or 0 causes the printing of a table of input dependent variable values (Y), the corresponding values predicted by the final regression equation and the deviations and percent deviations. 1 suppresses this calculation and printing.
IFCNST	62	I	Blank or 0 allows B_0 in (1) to be computed. 1 forces B_0 to have the value zero and suppresses the computation of the residual sums adjusted about the means.
IFTERM	64	I	Blank or 0 causes printing of table of input data values, 1 suppresses this printing.
FORMAT	65 - 66	I	Number of format cards to immediately follow this card. If blank or 0 or negative, 1 format card is assumed. Otherwise, format .LE. 10 with the value punched in 65 - 66.

F LEVEL AND SIGNIFICANCE

In order to control the likelihood of committing an error in entering a variable into the prediction equation when it is insignificant, or the error in removing a variable from the equation when it is significant, the corresponding F levels (FLVLIN and FOUT) must be specified by the user. The appropriate procedure to be followed is given below.

(1) Choose a maximum allowable likelihood for committing the error (e.g., .05). Call this value P.

(2) Calculate the probable final number of degrees of freedom NDF for analysis

$$NDF = N - K - 2$$

where N is the number of data sets (observations)
K is the number of variables finally entered in equation. Conservatively, K may be taken as the number of independent variables.

(3) Use Table 1, find the entry corresponding to the given P and NDF. If the given P and NDF are not listed, the user may interpolate or consult more detailed F tables in the various statistical tables.

TABLE 1: F LEVELS

NDF	P= .99	P= .95	P= .90	P= .50	P= .10	P= .05	P= .025	P= .01	P= .001
2	.00020	.0050	.020	.667	8.53	18.5	38.5	98.5	998.
3	.00019	.0046	.019	.585	5.54	10.1	17.4	34.1	167.
4	.00018	.0044	.018	.549	4.54	7.71	12.2	21.2	74.1
5	.00017	.0043	.017	.528	4.06	6.61	10.0	16.3	47.2
6	.00017	.0043	.017	.515	3.78	5.99	8.81	13.7	35.5
7	.00017	.0042	.017	.506	3.59	5.59	8.07	12.2	29.2
8	.00017	.0042	.017	.499	3.46	5.32	7.57	11.3	25.4
9	.00017	.0042	.017	.494	3.36	5.12	7.21	10.6	22.9
0	.00017	.0042	.017	.490	3.28	4.96	6.94	10.0	21.0
1	.00016	.0041	.017	.486	3.23	4.84	6.72	9.65	19.7
2	.00016	.0041	.016	.484	3.18	4.75	6.55	9.33	18.6
5	.00016	.0041	.016	.478	3.07	4.54	6.20	8.68	16.6
0	.00016	.0040	.016	.472	2.97	4.35	5.87	8.10	14.8
4	.00016	.0040	.016	.469	2.93	4.26	5.72	7.82	14.0
0	.00016	.0040	.016	.466	2.88	4.17	5.57	7.56	13.3
0	.00016	.0040	.016	.463	2.84	4.08	5.42	7.31	12.6
0	.00016	.0040	.016	.461	2.79	4.00	5.29	7.08	12.0
0	.00016	.0039	.016	.458	2.75	3.92	5.15	6.85	11.4
	.00016	.0039	.016	.455	2.71	3.84	5.02	6.63	10.8

The user should note that the likelihood of committing the converse error (e.g., failing to enter a variable in the equation that is, in fact, significant) increases as the likelihood of the first type of error decreases. The values corresponding to the .05 level are commonly employed.

The restriction is imposed that both F levels should be greater than or equal to zero, and that the F level for entering variables be at least as large as the F level for removing variables.

FORM OF INPUT DATA

The only restrictions on the form of input data are:

(1) The first item (the leftmost item read of the first card of each observation set) must be used as identification (an observation, or case number) for this data set. This means that this item must be positive (greater than zero) for all actual data sets, and this item is not included (for analysis) among the independent variables.

(2) The dependent variable must be the last one read by the input format (i.e., the rightmost variable read on the last card of the data set.) If the data is not in the prescribed order, it is not difficult to write a special ZFNCT subroutine to rearrange it. See REGRESSION ANALYSIS WITH TRANSFORMED DATA sub-section.

(3) If the data is weighted individually (i.e., the variable IFWT = 0), the weight must be the last item read in the entire data set.

(4) Following the last actual data set, the user must supply a complete blank data set (e.g., if a real data set occupies 1 card, supply 1 blank card, if a real data set takes 2 cards, supply 2 blanks, and so on).

Of course, the user must establish a consistent format for all the data sets. That is, each variable must appear in the same field in each data set. For example, if variable 10 occurs in columns 43-45 in the first data set, it must appear in 43-45 in all data sets.

After the data has been prepared according to the user's requirements, a format card (or cards) is prepared to describe the layout of the data to the program. Since each problem is preceded by its own control card and associated format, many different problems may be processed in one approach to the computer.

FORMAT CARD

The data is described to the program, variable by variable, field by field, by writing an appropriate format. Every item, including the observation number, the independent variables, the dependent variable and the weight (if any) must be described in the format. Further, every item is regarded as a floating point number. Thus, only E fields or F fields are allowed. Unused columns may be skipped (thus allowing a single data deck to be prepared for several problem runs) by using either the FORTRAN skip or the MAD skip. The user should become familiar with the various methods of writing formats in the FORTRAN or MAD manuals. The formats used by the regression procedure must begin with a left parenthesis ''('' and terminate with a right parenthesis '')''. The terminal * (asterisk) used in MAD formats is omitted in formats written for the regression program.

FORMAT CARD - Continued

In between the parentheses, F fields and skip fields, separated by commas, describe the data.

An F field is written
N F W.D where

N - No. of variables (fields) of the same size and decimal places.
If 1, this may be omitted.

F - The alphabetic character F.

W - The width of the field (i.e., the number of columns used for the variable).

. - The decimal point (must be included in the count ''W'', if it is punched in the data).

D - The number of places to the right of the decimal point. Used whenever no decimal point has been punched in the data.

A skip field is written

NX or SN where
N - is the number of columns to be skipped.
X or S- The alphabetic character X or S.

If each data set uses more than one card, punch a slash ''/'' in the format where the reading of the next card is to begin. In general, one should not end a format with / unless he is very familiar with the behavior of the input / output routines.

EXAMPLES:

1. Suppose each data set consists of 2 cards punched as follows.

1st Card

Col. 1 - 4: Data identification.

Col. 5 - 34: Var. 1 - 6, each 5 digits with 4 decimal places.

Col. 35 - 36: To be ignored in the analysis.

Col. 37 - 39: Var. 7, 3 digits with no decimal places.

Col. 40 - 55: To be ignored in analysis.

Col. 56 - 70: Var. 8 - 12, each 3 digits with 1 decimal place.

Col. 73 - 80: Card identification (To be ignored).

EXAMPLES: (Continued)

2nd CARD

Col. 1 - 16: Var. 13 - 20, each 2 digits with no decimal places.

Col. 17 - 20: Var. 21 (Dependent variable) 4 digits with 2 decimal places.

Col. 73 - 80: Card identification (To be ignored).

The FORMAT would be written

(F4.0,6F5.4, S2, F3.0, 16X, 5F3.1/8F2.0,F4.2)

2. Suppose each card is punched with the data items occupying 10 column fields, 7 per card. The format might be

(7F10.0)

If any of the data required places to the right of the decimal, punching the decimal point in the proper place in the number on the card will override the format specified for that number.

If all of the data could be contained on one card, there is some reduction in execution time that may be obtained by shifting the variables onto one card. If 2 cards are required as in example 1, there is no advantage in shifting the data around.

In general, the user should not use the columns beyond 72 for data. If more columns are needed, use additional cards. Note that use of the X and S FORMATS may be used to cause an effective rearrangement of the data without repunching the cards.

II. REGRESSION ANALYSIS WITH TRANSFORMED DATA

Sometimes the representation of the data may be greatly improved by applying the regression analysis to transformations of the raw data. In other words, a regression analysis will yield the best results when applied to the correct model for the phenomena. Failure to achieve a good fit may, thus, be due to either 1) Errors in the data, or 2) Errors in the model. The user may supply a transformation function for his data by writing and compiling a suitable subroutine. The subroutine, together with any required specification cards, follows the $ REGRESSION, EXECUTE card.

The subroutines are written in the MAD language or equivalent language and must conform to certain minor conventions. Also, a slight change in the control card is needed to signal the need to use the transformation subroutine.

These details are given below.

CONTROL CARD CHANGES

The control card preparation follows the description given earlier with 3 changes.

1. The variable NOVAR, in Cols. 36-40, is now the integer number of transformed variables (including the dependent variable, which may also be a transformed variable, if desired).

2. A new variable NOX is punched in Cols. 67-70, which is the integer number of raw data variables supplied (including the raw data dependent variable).

3. The variable IFTERM, in Cols, 64, causes the printing of the transformed data values if blank or 0. 1 suppresses this printing.

For example, suppose that 3 variables are supplied as raw data and that transformations are desired as follows:

$$Z_1 = X_1$$

$$Z_2 = X_1$$

$$Z_3 = X_1^3$$

$$Z_4 = X_2$$

$$Z_5 = X_2^2$$

$$Z_6 = X_2^3$$

$$Z_7 = X_3$$

In this case NOVAR = 7, NOX = 3.

TRANSFORMATION SUBROUTINE ZFNCT

The transformation ZFNCT may be written in MAD or an equivalent language, as the user desires. An example written in MAD will be given.

The argument list supplied by the calling program is as follows.

OBSNO - The observation (case) number of current set of raw data.

X - The base element of the raw data vector. If the subroutine is written in FORTRAN, this is the location of X(1). In MAD, this is the location of X(0).

XNUM - The number of raw data entries in the X vector. (In FORTRAN, the entries lie in the X vector from X(1) thru X(XNUM) - In MAD, the entries lie in the X vector from X(0) thru X(XNUM-1).)

Z - The base element of the transformed data vector. If the subroutine is written in FORTRAN, this is the location of Z(1). In MAD, this is the location of Z(0).

ZNUM - The number of transformed data entries in the Z vector. (In FORTRAN, the entries lie in the Z vector from Z(1) thru Z(ZNUM). In MAD, the entries lie in the Z vector from Z(0) thru Z(ZNUM-1).)

FIFWT - A variable whose value is non-zero if each data set has an associated weight read in as part of data. The variable is zero if each data set is assumed to have unit weight.

WHT - The weight associated with the current data set.

FIFERR - A variable whose value is to be set to
1) 0., if the subroutine has experienced no errors in the transformation of the data.
2) 1., if the subroutine has experienced an error that must terminate processing of the problem.
3) -1., if the subroutine has experienced a minor error that can allow the processing to proceed. A comment will be printed to flag this type of error.

It is to be noted that all of these parameters are floating point variables. Thus, if the subroutine is written in FORTRAN, some of these variables must be converted, in the subroutine, to FORTRAN integers in order to be used in computing subscripts.

The writing of ZFNCT subroutines may be clarified by means of the following example.

Suppose that 3 raw data variables are supplied and that the $\log_e(X_3)$ is to be the dependent variable. Further, each of the other independent variables are to be transformed according to the rules:

$$T_1 = X_1$$
$$T_2 = X_1^2$$
$$T_3 = X_1^3$$

The MAD subroutine to perform this set of transformations may be written as follows.

```
$ COMPILE MAD, PUNCH OBJECT

          EXTERNAL FUNCTION (OBSNO, X, XNUM, Z, ZNUM, FIFWT, WHT, FIFERR)

          ENTRY TO ZFNCT.

          FIFERR = 0.

          ZNUM = 0.

          THROUGH LOOP, FOR I = 0., 1., I.GE.XNUM

          THROUGH LOOP, FOR J = 1, 1, J .G.3

          INTEGER J

          Z(ZNUM) = X(I).P. J

LOOP      ZNUM = ZNUM + 1

          WHENEVER X(XNUM).LE.0.

          FIFERR = 1.

          OTHERWISE

          Z(ZNUM) = ELOG.(X(XNUM))

          END OF CONDITIONAL

          FUNCTION RETURN

          END OF FUNCTION
```

It should be apparent that the use of the ZFNCT subroutine allows the user complete freedom in the transformation (sometimes called ''EDITING'') of the data presented for analysis.

REPEATED USE OF DATA

The need to make several analyses of the same basic set of data sometimes arises (e.g., using the skip to select one of several dependent variables punched in the basic data set). One way to accomplish this would consist of reproducing the basic data deck as many times as is required, and supplying the multiple copies as indicated in the previous sections. Another way that avoids the reproduction of the data is the use of a special variable, IFSAVE, punched in columns 71-72 of the control card. This variable function as follows.

1. If the value of IFSAVE is blank or 0, the regression analysis program expects the data to follow the format card(s) in the usual way. (This is the normal situation.)

2. If the value of IFSAVE is -1, the regression analysis program expects the data to follow the format card(s) as usual except that a card with an asterisk ''*'' in column 1 and the words ''END OF PROBLEM'' in cols. 2-72 must follow the complete data deck, including the blank data set, but precede the next problem. In this case, a copy of the data is written (In decimal mode, as card images) on logical tape 4 for repeated use. Then tape 4 will be rewound and analysed according to the current controls and format.

3. If the value of IFSAVE is +1, the regression analysis program will take the data from logical tape 4 after reading only the appropriate format card(s) from the input tape. (Of course, tape 4 must have been written earlier by an analysis which had IFSAVE = -1 in the control card. Otherwise, the information on tape 4 is completely meaningless.)

Each analysis must find a control card and format card(s) on the input tape (as a part of the data deck supplied by the user) whether the data is to be read from tape 4 or from the input tape.

Each time the data is read, whether from the input tape or from tape 4 it will be interpreted according to the format card(s) immediately preceding.

It should be noted that data can be processed with and without transformations by using this feature. There is no present way to apply more than one ZFNCT subroutine during one approach to the computer. However, if any ZFNCT is included before the $DATA card, this (and only this) subroutine applies for this entire approach, and may be either utilized or ignored when processing the data.

REGRESSION ANALYSIS PROGRAM AS A SUBROUTINE

For very special purposes, it is possible to imbed the regression analysis program within any other program by calling it from the library like any other subroutine. In this case, $ EXECUTE is used, but $ REGRESSION is not used.

This may be accomplished by including one of the following calling sequences in the user's main program.

Main program in MAD,
EXECUTE REGRSN.

Main program in UMAP
CALL REGRSN

The user should be aware that, in the usual case, there is no return of control to the user's program once REGRSN has been entered.

However, by punching a 1 for the value of IFRTN (see control card) the user will obtain a return to his calling program after the end of the current problem. At that time, tapes 3, 4 and 9 will have been rewound. Tape 9 will contain the following information as one record.

NOIN The number of terms inserted in the regression equation (FORTRAN integer).

COEN The regression coefficient vector (101 locations floating point)

INDEX The term index vector (101 locations FORTRAN integers)

CONST The value of the constant term in regression equation.

Moreover, if the predicted values are asked for (i.e., IFPRED is blank or 0), then tape 3 will be written on during the solution. Also, if IFSAVE is used to save the raw data, then tape 4 will be written during the solution.

The following MAD sequence will illustrate use of the regression equation (e.g., for plotting graphs, etc.) to determine the predicted value YPRED for a set of X values.

```
      DIMENSION X(101), COEN (101), INDEX (101)
      EXECUTE RTAPE9.  NOIN, COEN, INDEX, CNST)
      INTEGER NOIN, INDEX, I
      YPRED = CONST
      THROUGH LOOP, FOR I = 1, 1, 2 .G. NOIN
LOOP  YPRED = YPRED + COEN(I)*X(INDEX(I))
```

The subroutine RTAPE9 takes care of the reading of the information placed on tape 9 by REGRSN and the conversion of FORTRAN integers to MAD integers.

The subroutine REGRSN is fairly large and calls upon several other routines. Thus, the user desiring to use REGRSN as a subroutine should not plan on using more than approximately 20000 octal (8192, decimal) locations in the calling program. The exact allowable size depends on the subroutines called by REGRSN and the main program. If the size becomes too large to be loaded in a single core load, the user is referred to the description of the PING-PONG method of subdividing large programs.

EXAMPLES

1) Assume there is a subroutine on the MAMOS library which computes Y = F(X). Write a program which reads X, computes Y, and prints X and Y. Suppose the subroutine name to be MOREV9. Both X and Y are in PROGRAM COMMON.

```
$EXECUTE        MAMOS
$ID    PINGPONGER*XXX/YY/ZZZ*5MINUTES*10P$
          R PROGRAM TO USE MOREV9.
            PROGRAM COMMON X,Y
START       READ FORMAT INPUTS,X
            EXECUTE MOREV9.
            PRINT FORMAT OUTPUT,X,Y
            TRANSFER TO START
            VECTOR VALUES INPUTS =$25X,F10.4*$
            VECTOR VALUES OUTPUT =$3H1X=F10.4,3X,2HY=E16.8*$
            END OF PROGRAM
$DATA
BEST VALUE OF PI HANDY IS 3.14159
```

2) Suppose example 1) above will not work because MOREV9 and the input-output routines combined require a little too much storage for execution. A PING PONG job can be done so that input-output routines are not needed at the time MOREV9 is used.

```
$EXECUTE        MAMOS
$ID    PINGPONGER*XXX/YY/ZZZ*5M*10PAGES$
```

```
$ COMPILE MAD, EXECUTE
          R CORELOAD 1--GOES TO TAPE 2, RECORD 1.
            PROGRAM COMMON X,Y
            READ FORMAT INPUTS, X                          [CORELOAD]1
            SELPGM.(1,4)
            VECTOR VALUES INPUTS =$25X,F10.4*$
            END OF PROGRAM
$BREAK
```

```
$COMPILE MAD
          R CORELOAD 2--GOES TO TAPE 4, RECORD 1.
            PROGRAM COMMON X,Y
            EXECUTE MOREV9.                                [CORELOAD]2
            SELPGM.(2,4)
            END OF PROGRAM
$BREAK(4)
```

```
$COMPILE MAD
          R CORELOAD 3--GOES TO TAPE 4, RECORD 2.
            PROGRAM COMMON X,Y
            PRINT FORMAT OTPUT,X,Y                         [CORELOAD]3
            SELPGM.(1,2)
            VECTOR VALUES OTPUT =$3H1X=F10.4,3X,2HY=E16.8*$
            END OF PROGRAM
$BREAK(4)
$DATA
BEST VALUE OF PI HANDY IS 3.14159
```

3.6 MAMOS Organization and Coding Information

MAMOS consists of a low core package (IOS) of many subroutines (including I/O subroutines), a monitor and loader, the MAD compiler, the UMAP assembler, the ALGOL compiler, the MADTRAN translator, a subroutine library, and several other records.

The low core package remains in core at all times when MAMOS has control. Its logical function as part of the operating system is to provide end of job processing such as requested dumps, and to call in system records. The low core package occupies cells 2048 through 4095 so object programs have cells 4096 through 32767 available, unless an installation desires to reserve part of upper memory.

All I/O functions under MAMOS are accomplished through use of IOEX.

A job is processed under MAMOS in the following manner:

a) The low core package reads in the monitor-loader-accounting record and gives control to the monitor.

b) The monitor does any necessary actions to terminate a previous job which may not be signed off. The beginning of the job is found and the job is signed on.

c) Mamos control cards are scanned and when a translator is necessary the first record for that translator is read into core and control is given to the translator. The translator carries out its function and returns to the low core package.

d) The low core package again reads in and gives control to the monitor record.
Steps c) and d) are repeated until a $BINARY card, a $DATA card or an end of file is detected. During these steps and if execution is legal, any binary output from translation is stacked on the execution tape (logical tape 3). Binary cards contained in the job deck are simply transfered to the execution tape if they are not preceded by the $BINARY control card.

e) When a $DATA card, a $BINARY card or an end of file is detected, a check is made to see if execution is legal. If execution is not legal then the job is terminated. Execution is legal if it was requested and if no error was detected by the monitor and translators as the job deck was processed.

f) If execution is legal, control is passed to the loader which loads any binary program decks which may have followed a $BINARY card, then the execution tape is loaded, and finally any other routines which are necessary and available from the library.

g) Execution of the program begins. Execution is usually terminated by calling SYSTEM or ERROR subroutines in the low core package, or by trying to read more data than was included in the job deck.

Low Core Package (IOS)

IOS consists of many subroutines which are used primarily by MAMOS executive routines and the library routines to accomplish input-output. A stripped down version of IOS could be useful for other applications under other monitors.

There are two I/O unit tables in IOS, and storage is set aside for saving an I/O table temporarily. The master I/O unit table consists of the standard I/O unit definitions, and the working I/O unit table consists of the current unit definitions as the job is processed.

Both unit tables consist of entries of the form PZE SYSUNi or MZE SYSUNi for each logical number which is defined, where SYSUNi is a standard IBSYS unit name. A table of the current master I/O units is given below.

The working I/O unit table is usually identical to the master I/O unit table. However, the working table may be saved by a subroutine, the definitions may be altered in the working table, and when the desired I/O is completed, the saved table may be restored.

There are three subroutines which automatically do single record buffering. These routines are used almost exclusively for (1) reading the input tape (SYSIN1 = logical tape 7), (2) writing the output tape (SYSOU1 = logical tape 6), and (3) writing the punch tape (SYSPP1 = logical tape 5). These self-buffering routines may, with care, be used for reading/writing of other logical units by altering the working I/O unit table.

There is a fixed communication region in IOS starting at octal location 3720. Most of the values in this region are defined in the system symbol table of UMAP and are available to UMAP codes through use of the pseudo-operation SST. All of the low core subroutine entry points are included in the system symbol table.

Most of the subroutines of IOS are available to relocatable codes such as MAD programs through a library subroutine which connects relocatable calls to the low core subroutines.

LOGICAL I/O UNITS FOR MASTER TABLE

Logical Number	IBSYS UNIT	Use
1	SYSLB1	System
2	SYSUT1	Scratch
3	SYSUT2	Execution
4	SYSUT4	Intermediate 1
5	SYSPP1	Punch
6	SYSOU1	Output
7	SYSIN1	Input
8	SYSLB2	Library if desired
9	SYSUT3	Intermediate 2
10	SYSCK2	Available
11	SYSCK1	Available

Tapes 10 and 11 are not normally used by the system so with the library on SYSLB1 it is possible to operate MAMOS with a minumum of 8 tapes. However, best operation is achieved when the library is the first file on SYSLB2.

If desired, logical tape 5 may be assigned to SYSOU1 rather than to SYSPP1. In this case, the routine which prints SYSOU1 must handle mixed mode records. Print information is in BCD mode and punch information is in binary mode.

SUBROUTINES FOR NON-DATA SELECTS

The following routines are used for non-data reference to logical tape numbers 2, 3, 4, 9, 10, and 11. They may be used to refer to logical numbers 1, 5, 6, 7, and 8 but before doing so, a special cell in IOS must be set non-zero for each reference. This cell is called SOK567 and is in octal location 4040. Reference to illegal logical numbers results in the printing of a message and termination of the job.

In the following calling sequences N is the logical tape number, and except when specified differently, return is always to the second instruction following the TSX.

Calling sequences

1) Rewind tape N
```
    TSX    REWTAP,4
    TIX    0,0,N
```
2) Backspace tape N one record
```
    TSX    BSRTAP,4
    TIX    0,0,N
```
3) Rewind and unload tape N
```
    TSX    RUNTAP,4
    TIX    0,0,N
```
4) Backspace tape N one file
```
    TSX    BSFTAP,4
    TIX    0,0,N
```
5) Write end of file on tape N
```
    TSX    WEFTAP,4
    TIX    0,0,N
```
6) Set tape N to low density
```
    TSX    SETLOW,4
    TIX    0,0,N
```
7) Set tape N to high density
```
    TSX    SETHIH,4
    TIX    0,0,N
```
8) Skip M records on tape N
```
    TSX    SKPREC,4
    TIX    M,0,N
```
9) Skip M files on tape N
```
    TSX    SKPFIL,4
    TIX    M,0,N
```
10) Check activity of tape N
```
  L TSX    CHEKIO,4
L+1 TIX    T,0,N
```

If T=0 then control will be returned to L+2 only after tape N is inactive.

If T≠0 and tape N is inactive then control goes to T.

If T≠0 and tape N is active then control is immediately returned to L+2.

Note: Skipping of files and records is overlapped, so computing (and I/O on channels different than the one which N is on) may go on while the skipping is done.

DATA SELECT SUBROUTINE

Except for some system record reading and non-data selects, all I/O is accomplished under MAMOS through use of a select routine with four entries. The four entries are for read and write in both BCD and binary modes.

The routine is quite useful for programs which require special input output. Also, I/O buffering routines may be easily written through use of this select routine. An important feature of the routine is that it may be called at trap time.

The input output is accomplished exclusively through use of IOEX, but the user need not know IOEX.

Calling Sequence

```
TSX  XXXXXX,4
TIX  EOR,0,N
TIX  L(IOC),W,ETT
TIX  EOF,T,RTT
Return
```

XXXXXX = RDSBIN for reading binary records.
= RDSDEC for reading BCD records.
= WRSBIN for writing binary records.
= WRSDEC for writing BCD records.

Use

N = the logical tape number to be read or written.
W≠0 if it is desired to wait until the I/O operation is completed before returning to the caller.
T≠0 if only one try is desired for reading even though the record may be redundant.
EOR, ETT, EOF, and RTT are trap time exits to the user's routines. Any or all of the exits may be zero.

A user's exit must be to a routine which may set switches etc., and then return by means of a TRA 1,4.

L(IOC) = the location of the first of a block of I/O commands. Up to 10 commands are allowed. If more than 10 commands are necessary then at least one of the first eleven must be a TCH command.

The I/O commands must terminate with a command which causes a channel interrupt, i.e. the last command must be a IOXT.

The first ten of the I/O commands are moved to storage within the select routine so the original block at L(IOC) may be modified immediately upon return from the select routine.

Noise records as defined in IOEX will be accepted if there is at least one input output and proceed command preceding the last I/O command, i.e. the IOXT. Hence, tape may be erased by the two I/O commands.

```
IOCP  0,0,0   (IOBP in UMAP)
IORT  0,0,0   (IOBT in UMAP)
```

However, if only the second of the above two commands was used then there would be a noise indication if writing.

Users Exits

EOR, ETT, EOF, and RTT, if non-zero specify entries to subroutines coded by the user. Each of these subroutines must carry out its desired function and return by means of a TRA 1,4.

An entry to a user's routine is made at trap time, i.e. when an interrupt condition occurs due to channel command trap, a redundant read or write, detection of an end of file in reading or detection of the end of tape in writing.

On entry to a user's routine the following information is available.

a) The address of the accumulator contains the number of words read or written by the channel command just completed or in use at time of interrupt.

b) The decrement of the accumulator contains the logical number of the unit in use at time of interrupt.

c) Index register 2 contains the channel number of the channel which causes the interrupt. Channel A = 1 and Channel B = 2.

d) Index register 1 contains the two's compliment of the address of the cell which has the result of a store channel instruction at time of interrupt.

Restrictions on the User's Routine

1) The user's routine must exit by means of a TRA 1,4.

2) For efficient I/O the user's routine should not be time consuming.

3) Only one of the user's routines is entered for a single trap. The order of checking for an exit is as follows:

READING	WRITING
End of file exit (EOF)	End of tape exit (ETT)
Redundancy exit (RTT)	Redundancy exit (RTT)
End of record exit (EOR)	End of record exit (EOR)

4) If no ETT exit is supplied for writing and the end of tape is encountered, then 2 end of files are written on the tape, the tape is rewound and unloaded, an on-line message is printed for the operator and the machine pauses for a fresh tape. Then a check for RTT and EOR exits will be made. None of the above actions are taken if there is an ETT exit.

5) Activity checking by calling CHEKIO is permissible only if in the sequence.

```
TSX   CHEKIO,4
TIX   T,0,N
```

T is non-zero. If T is zero, and logical unit N is active, then an endless wait will occur.

6) Index registers 1 and 2, the AC, MQ, and indicators need not be saved by the user's routine.

7) Calls to REWTAP, BSRTAP, RUNTAP, BSFTAP, WEFTAP, SETLOW, SETHIH, SKPREC, SKPFIL, RDSBIN, RDSDEC, WRSBIN, and WRSDEC may be issued by a user's trap time routine, but only for a unit on the same channel on which the trap occurred.

8) Storage is allocated for several blocks of I/O commands and parameters. One of these blocks is reserved whenever a logical unit is active. It is possible (if there is not one block per logical unit) that a block will not be available when activity is required. A data select at non-trap time causes no trouble because an automatic wait for a free block will occur. However, at trap time there may not be more than one block available and no more than one data select should be issued without insuring there is an available block.

Restrictions 7) and 8) above may be overcome by means of a special trap time routine which may be called by the user's routine. The calling sequence is as follows:

```
L    TSX   ISITOK,4
L+1  TIX   BUSY,0,N
```

Control will return to L+2 if it is permissible to select logical unit N. Control is returned to location BUSY if (1) logical unit N is on a channel different from the one for which the trap occurred or (2) logical unit N is busy or (3) there is no storage block for I/O commands.

On entry to ISITOK, it is assumed that the accumulator contains what it had at the time the user's routine was entered, since the logical unit number in the decrement of the accumulator is used in determining if the channel which N is on is the same as the one for which the trap occurred.

If ISITOK is to be entered more than once, then the second or greater entry may be made to ISIT11 rather than ISITOK and the accumulator as saved on the first call will be used.

It is always permissible to re-select the logical unit for which the trap occurs, and for this type of use there is no requirement to call ISITOK.

The restriction on I/O command storage blocks could be completely removed by allocating (by assembly parameter) one block per defined logical unit, but since IOS is limited in its available storage there can only be from 5 to 8 blocks. However, it is seldom that more than 5 units are in use at any one time.

READ INPUT TAPE SUBROUTINE

This subroutine has two entries and is used for reading of the system input tape SYSIN1, which is logical tape 7 under MAMOS.

Calling sequences

```
L     TSX    SCARDS,4    or    TSX    SPEEK,4
L+1   TIX    A,0,EOF           TIX    A,0,EOF
L+2   Return                   Return
```

Use

Entry to SCARDS causes the next record on logical tape 7 to be stored in locations A, A+1,..., A+j-1 where j=14 if the record is BCD and j=28 if the record is binary. Also, filling of the buffer is initiated and then control returns to L+2. On return, the AC will be zero if the record following the one just transmitted is BCD. The address portion of the MQ will contain 14 if the record transmitted was BCD and will contain 28 if the record transmitted was binary.

Entry to SPEEK is the same as to SCARDS except the initiation of filling the buffer is omitted. Hence, one may ''take a look'' at an input record before reading it.

If the next record on the input tape is an end of file and EOF=0 then a message is printed and the job is terminated. If EOF≠0 and an end of file is detected then control is sent to location EOF. Handling of end of file exits is the same for both entries SCARDS and SPEEK.

Single record buffering is automatically started on the first call, and also when the subroutine is called after an end of file is read.

The subroutine will not handle blocked input, and expects look ahead words as follows:

Word 14 of a BCD record $=(XXXX60606060)_8$ or $(XXXX00000000)_8$ if the next record is BCD.

Word 28 of a Binary record $=(000000010000)_8$ if the next record is BCD.

If SPEEK is called, the look ahead bits are transmitted with the record. If SCARDS is called then look ahead bits in BCD records are replaced by blanks, and look ahead bits in binary records are replaced by zeros.

Noise records are ignored. Records which are permanently redundant are accepted as read the last time. The number of tries before calling a record permanently redundant is an assembly parameter of IBSYS.

WRITE OUTPUT TAPE SUBROUTINE

This subroutine is used for writing of the system output tape SYSOU1 which is logical tape 6 under MAMOS. It does single record buffering of BCD records of length no greater than 22 words.

Calling sequence

```
L     TSX    SPRINT,4
L+1   TIX    A,0,K
L+2   Return
```

Use

Entry to SPRINT causes K BCD words (or 22 words if K > 22) to be written in BCD mode on the output tape SYSOU1 as one record. The words are taken from locations A, A+1,..., A+K-1 and transferred to an output buffer for printing. Control returns to L+2.

During execution, SPRINT examines the first character in order to keep track of the number of pages being printed.

The following characters have meaning for the purpose of page counting.

Character	Meaning
1	Skip to new page.
2	Skip to middle of a page.
4	Skip to next fourth of a page.
6	Skip to next sixth of a page.
8	Same as 6.
+	Suppress space.
blank, or 9	Single space.
0	Double space.
-	Triple space.

All other characters are treated as blank by SPRINT in counting of pages.

If the standard, or estimated page count is exceeded during execution, then a message is printed and the job is terminated. A page is considered as 60 printed or blank lines.

WRITE PUNCH TAPE SUBROUTINE

This subroutine has two entries and is used for all writing of the system punch tape SYSPP1, which is logical tape 5 under MAMOS. It does single record buffering of binary records, 28 words per record. BCD information is rotated to binary before writing it on the punch tape as a binary record.

Calling sequence

```
L     TSX    SPUNCH,4     or    TSX    DPUNCH,4
L+1   TIX    A,0,K              TIX    A,0,K
L+2   Return                    Return
```

Use

Entry to SPUNCH causes K binary words + (28-K) zero words (or 28 if K>28) to be written on the punch tape SYSPP1 as a 28 word binary record. The words are taken from locations A, A+1,..., A+K-1 and transfered to an output buffer for printing. Control returns to L+2.

Entry to DPUNCH causes K BCD words + (14-K) blank words (or 14 if K>14) to be rotated to column binary form as 28 words. These 28 words are then written on the punch tape SYSPP1 as one binary record.

If, during execution, the standard or estimated punched card count is exceeded, then a message is printed and the job is terminated.

SUBROUTINES USED WITH BUFFERING ROUTINES

The following routines are used in conjunction with the buffering subroutines SCARDS, SPEEK, SPRINT, SPUNCH, and DPUNCH.

Calling sequences

```
TSX    ENDCDS,4    or    TSX    ENDPNT,4    or    TSX    ENDPCH,4
Return                   Return                   Return
```

The above entries cause unbuffering of logical tape 7 or 6 or 5. These entries must be used before switching logical units associated with the buffering routines.

```
TSX    REWCDS,4    or    TSX    REWPNT,4
Return                   Return
```

The above entries cause logical tape 7 or 6 to be unbuffered and rewound.

```
TSX    BSRCDS,4
Return
```

The above entry causes logical tape 7 to be unbuffered and backspaced one record.

```
TSX    BSFCDS,4
Return
```

The above entry causes logical tape 7 to be unbuffered and backspaced one file.

```
TSX    NXFCDS,4
Return
```

The above entry causes logical tape 7 to be unbuffered and one file to be skipped.

```
TSX    WEFPNT,4
Return
```

The above entry causes logical tape 6 to be unbuffered and an end of file to be written on logical tape 6.

I/O UNIT TABLE SUBROUTINES

The following subroutines are used for saving, restoring, altering, and initializing the working I/O unit table.

Calling sequences

1) Save the working I/O unit table.

```
TSX     SAVTBL,4
Return
```

2) Restore the previously saved I/O unit table.

```
TSX     RETTBL,4
Return
```

3) Alter the working I/O unit table.

```
TSX     SETTAP,4
TIX     M,0,W
Return
```

A delay occurs until logical unit W becomes inactive, then logical unit W is altered to become the actual unit associated with logical unit M of the master I/O unit table.

4) Initialize the working I/O unit table.

```
TSX     ORGTBL,4
Return
```

The working I/O unit table is replaced by a copy of the master I/O unit table.

5) Delete logical unit N from working I/O table.

```
TSX     VOID,4
TIX     0,0,N
Return
```

This subroutine will cause logical unit N of the working table to be illegal for use until its restoration by one of the above entries.

Example: Suppose during execution, it is desired to read logical tape 9 using the buffering routine SCARDS.

The following code would terminate buffering on the regular input tape and alter the I/O table. Then after logical 9 had been read, the I/O table would be restored to its previous condition.

```
        TSX     SAVTBL,4        SAVE PRESENT TABLE
        TSX     ENDCDS,4        UNBUFFER THE INPUT TAPE
        TSX     REWTAP,4        REWIND THE NEW INPUT TAPE
        TIX     0,0,9
        TSX     SETTAP,4        ALTER TABLE FOR NEW INPUT TAPE
        TIX     9,0,7
*     CODE USING LOGICAL 9 OF MASTER AS THE INPUT TAPE
*  WHICH IS READ BY THE SUBROUTINE SCARDS.
          .
          .
          .
        TSX     ENDCDS,4        UNBUFFER THE NEW INPUT TAPE
        TSX     RETTBL,4        RESTORE TO PREVIOUS TABLE
```

OCTAL CORE DUMP SUBROUTINE

This subroutine has two entries and is used for taking octal dumps of all or part of core storage.

Calling Sequences

```
L     TSX    SDUMP,4     or     TSX    CDUMP,4
L+1   TIX    A,0,B              TIX    A,0,B
L+2   Return                    Return
```

The above entries cause the core storage block A,A+1,...,B to be printed in octal, 8 cells per line. If all 8 words for a line are equal and also equal to the 8th word of the previous line, then this line is not printed. Instead, one line of periods (.) is printed for each group of consecutive lines whose words are all equal.

If the SDUMP entry is used then each printed line also has the octal address of the first word printed at the beginning of the line.

If the CDUMP entry is used then an octal address is printed as for SDUMP except that the address is relative to 1. That is, the addresses printed will be the octal equivalent of 1,8,16,....

SYSTEM RECORD READING SUBROUTINE

This subroutine is used for reading of all system records. Either the select routine (described previously) or SYSLDR of IBSYS is used in the reading.

Calling Sequences

```
TSX    SELRCD,4
TIX    ID,0,N
```

Entry to SELRCD causes the record with identification ID to be found on logical tape N, the record is read into core and control is given to the specified entry point (ENTRY) of the record. If ID = 0 then the next system record on logical tape N is read rather than searching for a record with matching identification.

Restrictions

Logical tape N must be positioned within the file containing the desired system record, and if a backwards search is necessary then the file should end with a dummy system record which has an identification number of $(77777)_8$. System records within a file should have consecutive integers as identification so that backward searching may be done correctly.

System Records

The form is as follows:

Word	Contents		
1	IOCP	RECNMX,0,1	RECNMX = $(4061)_8$
2	BCI	1,NAME	
3	IOCP	RECIDN,0,1	RECIDN = $(4060)_8$
4	PZE	ID	
5	IOCP	SYSTRA,0,1	SYSTRA = $(100)_8$
6	TRA	ENTRY	
7	IOCP	A1,0,N1	
	[N1 words]		
	.		
	.		
	.		
	IOCP	AN,0,NN	
	[NN words]		
	IORT	0,0,0	

This type of system record is used for all MAMOS records on the IBSYS operating system tape(s).

ON-LINE PRINT SUBROUTINE

This subroutine is used to print on-line messages for the operators.

Calling sequence

```
TSX     ONLINE,4
TIX     A,0,M
Return
```

Entry to ONLINE causes M BCD words to be printed on the on-line printer, 12 words per line. The words are taken from storage locations A,A+1,...,A+M-1.

ONLINE uses the IBSYS subroutine (PROUT to do the actual printing.

PAUSE SUBROUTINE

All pauses under MAMOS are done by giving the instruction TSX .PAUSE,4.

.PAUSE goes to the IBSYS subroutine (PAUSE where the machine stop occurs. It is very important for all machine stops during execution to occur in (PAUSE so execution time will not be counted while the machine is halted.

FLOATING POINT TRAP ROUTINE

This routine has one entry (.FTRAP) and handles all floating point traps under MAMOS. Cell 8 is initialized with a transfer to the floating trap routine at the beginning of each job.

Overflow in any floating point operation is always considered fatal and the job is terminated.

Underflow in the least significant half of the result of a floating point operation is always set to zero and is not considered an error.

Underflow in the most significant half of the result of a floating point operation is either treated as a fatal error, or set to zero and ignored. The treatment of this condition depends on a switch which may be set by the programmer.

If it is desired to ignore high order underflow then a TSX FTRAP,4 should be given.

If it is desired to treat high order underflow as a fatal error, then a TSX NTRAP,4 should be given.

The routine at the beginning of the job is always set to treat high order underflow as a fatal error.

When an overflow/underflow is determined to be fatal, a message is printed describing the type of error, and the location of the floating point operation which caused the error, then the job is terminated.

TRACE ROUTINE

If the $SUBTRACE specification is used for a job, then the loader links all subroutines it loads to the low core routine STRACE.

There is an on-off switch in STRACE. If the switch is ''on'' and if $SUBTRACE was used then each call to a subroutine produces a line of printed output which gives the name of the subroutine called and the location from which it was called.

The switch in STRACE is set to ''on'' at the beginning of each job, but during execution the switch may be set ''on'' or ''off'' by means of library subroutines.

A few of the subroutine calls generated by the MAD compiler for arithmetic functions are not included in a subroutine trace print out.

LOGICAL TO ACTUAL UNIT SUBROUTINE

This subroutine gets the actual unit corresponding to a specified logical unit.

Calling sequence

```
TSX     GETNAM,4
TIX     0,0,N
Return
```

The actual unit which corresponds to logical unit N is converted to BCD and returned in the logical accumulator. For example, if the actual unit were A3 then the BCD word A30000 would be returned.

ERRONEOUS TRANSFER ROUTINE

Just before execution of a job is begun, all unused core is filled with the instruction TSX SCATCH,4.

An erroneous transfer to one of these cells causes a message to be printed and the job to be terminated.

The octal equivalent of this transfer instruction is 007400403771.

END OF JOB ENTRIES

All jobs terminate by going to the low core entry called SYSTEM. Entry to SYSTEM causes the current job to be terminated and the next job is then processed.

Another entry to low core which terminates a job is called ERROR and the only difference between ERROR and SYSTEM is that the ERROR entry causes any requested dumps to be taken before transfering to SYSTEM.

TRANSLATOR ENTRY

All translators terminate by sending control to the low core entry called TRANXT. Entry to TRANXT causes the monitor record to be read into core and control goes to the monitor with an indication that a translator has just completed its function.

3.7 THE UMAP ASSEMBLY PROGRAM UNDER MAMOS

UMAP under MAMOS is a modified version of UMAP (University of Michigan Assembly Program) developed by the University of Michigan. UMAP and FAP (under the FORTRAN II Monitor) are very similar and many programs written for UMAP would also be compatible with FAP.

Major modifications and additions to UMAP (as received from the University of Michigan) to produce the UMAP under MAMOS are as follows:

1) Blocked input capability so UMAP accepts blocked input of the type produced by the FAP assembler under FORTRAN II Monitor.

2) All 7090/7094 machine instructions, except those noted at the end of the section, were made available.

3) The 'EVEN' pseudo-operation was put into UMAP so the 7094 double precision instructions could be used.

4) Several pseudo-operations were put into UMAP to provide more compatibility between FAP and UMAP.

The programmer already familar with FAP may turn to the end of the section where differences between FAP and UMAP are given.

Most of the following description of UMAP under MAMOS is taken from the write-up of the University of Michigan Executive System.

INTRODUCTION

UMAP is an assembler, as opposed to a compiler such as 'MAD'. The exact meanings of such terms are difficult to state, but basically the difference between a compiler and an assembler is in the 'level' of the source language--the source language of an assembler is closely related to the computer command structure, whereas the source language of a compiler resembles the technical notation in which problems are stated by human beings. Inherent in this difference is the fact that while an assembly language provides a programmer with a maximum degree of flexibility in constructing a program, it also requires a rather complete understanding of the computer itself and its manner of operation.

A 7090/7094 machine-language program is a sequence of 36-bit binary numbers which represent both the machine instructions necessary to perform the desired task and the data to be operated upon. Working in such a language entails rather obvious hardships upon a programmer. For this reason, symbolic languages are usually used to communicate with a computer. A symbolic-language program is, then, merely a representation of a machine-language program in a form more convenient for human beings. Use of a symbolic language, however, requires that a translation from this language to machine-language be performed before a program may be run on a computer--this is the process of assembling or compiling.

Thus, UMAP accepts as input a program written in a specified symbolic language and produces the equivalent machine-language program as its output. The term 'UMAP' is used both as the name of the symbolic-language and as the name of the program which translates this symbolic-language into machine-language.

The purpose of this section is to describe the symbolic-language which the UMAP assembler will accept as input. It is assumed that the reader is familiar with such concepts as 'relocatable' program, 'PROGRAM COMMON' or 'ERASABLE' storage, and 'program card' which are not described here as far as their function in a program is concerned. This section describes only the manner for obtaining such quantities in a UMAP program.

PROCESSING BY UMAP

UMAP assumes that a program is relocatable and a main program. Either of these assumptions may be modified by pseudo-ops which are available in the language. Also, UMAP generates automatically a program card for the program and a transfer vector. These are needed for loading the program and linking it to other programs loaded with it. This generation may be stopped by the occurrence of any one of several pseudo-ops. The language consists of a set of operation codes plus rules on how to form the various entities needed. The operation codes may be divided into two main types--pseudo-ops and machine instructions. Pseudo-ops are operation codes which do not correspond to actual machine instructions. These may cause the generation of machine-words or may affect the processing of the remainder of the program. Pseudo-ops accepted by UMAP are described later in this section. Machine instructions are simply symbolic names for the actual operations built into the 7090/7094.

In translating a program, UMAP processes the source deck twice--PASS 1 and PASS 2. In PASS 1, all operation codes are analyzed and deciphered. The machine version of the operation code is preserved for PASS 2 processing. Almost all symbols are found and defined on PASS 1. See the sub-sections on symbol definition and PASS 2 symbol dfinition. The information for the program card (ERASABLE and PROGRAM COMMON storage, program length, number and name of all subroutines called, and the names and locations of all entry points) is gathered and the transfer vector is formed. Machine instructions and certain pseudo-ops are scanned completely on PASS 1 to find all literals, all occurences of the /TV/ qualifier, and, in the case of some pseudo-ops, to define the location field symbol. Those pseudo-ops which require processing on PASS 1 are examined and the appropriate action taken . Finally, each card is placed on an intermediate tape along with its deciphered operation code, flags pertaining to PASS 1 errors, and information obtained during PASS 1 processing.

Between passes, the program card is generated and the transfer vector is constructed. The symbols defined on PASS 1 are re-evaluated in terms of the length of program common and the transfer vector. The program card and transfer vector are printed to initiate the printed listing of the program.

In PASS 2 the program is read from the intermediate tape and final processing is performed. Each card is analyzed and appropriate action taken. If the card produces machine words, these are printed (in the octal number system) on the listing, along with the original card. Also, if execution of the program is expected, these binary words are placed on a tape to await the execution phase. If an object deck is requested, these binary words are placed on an output tape to be punched as binary cards. Some symbols are defined on PASS 2 if necessary. See the PASS 2 symbol definition sub-section. All symbol definitions are checked. If the PASS 1 and PASS 2 definitions don't compare an error flag (P flag) is produced on the listing. Whenever an error is found in processing, an appropriate error flag is printed in the listing along with the card. See the sub-section on error flags.

After PASS 2 processing is completed, certain additions are made to the listing. Multiply-defined and undefined symbols are listed, reference tables for all defined symbols and for all literals are printed, and an assembly statistics table is printed.

SYMBOLS

A symbol is a string of one to six non-blank characters, at least one of which is non-numeric, and none of which is among the following set of 'break' characters--

+	PLUS	-	MINUS
*	ASTERISK	/	SLASH
,	COMMA	=	EQUAL
(	LEFT PARENTHESIS	)	RIGHT PARENTHESIS

SYMBOLS (Continued)

Symbols are used as names for memory locations and program parameters. Due to the right justification of symbols (with leading zeroes) during UMAP processing, symbols may not start with a zero. For example, the following are legal symbols--

A

SYMBOL

12AB3

A.1

SYMBOL DEFINITION

Symbols are normally defined on PASS 1 of UMAP, but, in certain conditions, definition may occur on PASS 2. By 'definition' of a symbol is meant an assignment of a numerical value and a mode to the symbol. There are three possible modes for symbols--erasable, relocatable, and absolute. An absolute symbol is one which refers to a fixed number; one which is invariant to where, in memory, the program is located. An erasable symbol is one which is assigned to erasable storage through use of the 'ERAS' or 'ERLIST' pseudo-ops. Relocatable symbols are symbols whose values are dependent upon where, in memory, the program is located. These symbols always refer to storage locations in relocatable programs. In an absolute assembly, all symbols are absolute. Symbols are defined after they have occurred in the location field of a machine instruction, in the location field of certain pseudo-ops (e. g., 'CALL' and 'VFD'), after the /TV/ qualifier, as the name of a subroutine in a 'CALL' or 'CALLIO', or in the variable field of certain pseudo-ops (e. g., 'ZERO', 'ASSIGN', 'EXTERN', and 'ERLIST'). A symbol is normally defined once only in a given program.

PASS 2 SYMBOL DEFINITION

A certain limited amount of PASS 2 symbol definition may occur in UMAP in connection with the 'EQU', 'BOOL', 'SET', and 'SYN' pseudo-ops. Normally these pseudo-ops define symbols in their location fields as the equivalence of their variable field expressions on PASS 1. If, however, the variable field is undefined on PASS 1, then the location field symbol remains undefined until the card is encountered again on PASS 2. At this time, if the variable field is now defined, the location field symbol is defined. Note that this makes possible the situation in which a symbol is undefined for part of the assembly and defined for the rest of it. In connection with this, it should be noted that literals are undefined on PASS 1, so that if one writes

```
A       EQU     =15
```

A is undefined until this 'EQU' card is encountered on PASS 2.

ELEMENTS AND TERMS

An element is either a single integer less than 2^{35} or a single symbol, either possibly preceded by one or more qualifiers. An absolute, relocatable, or erasable symbol is regarded, respectively, as an absolute, relocatable, or erasable element. An '*' appearing as an element (not as an operator) has the meaning 'present location'. In a relocatable program, an '*' as an element is a relocatable element. In an absolute assembly, it is an absolute element. Thus, the statement

```
ALPHA    TRA     *+5
```

is the same as

```
ALPHA    TRA     ALPHA+5
```

An integer is always an absolute element.

A term is a string composed of elements and the operators

* for Multiplication and

/ for Division.

A term may consist of a single element, two elements separated by '*' or '/', three elements separated by two operators, etc. A term must begin with an element and end with an element. Two operators may not occur together, nor may two terms occur together. For example, the following are all terms.

```
ABC

A/3*C

4*BCK/13K/A13*K1XY
```

ARITHMETIC EXPRESSIONS

An arithmetic expression is a string composed of terms separated by the operators

\+ for Addition and

\- for Subtraction.

An expression may consist of a single term, of two terms separated by '+' or '-', of three terms separated by two operators, etc. Two operators may not occur together, nor may two terms occur together, but an expression may begin with an operator. No parenthesization is allowed in expressions. Examples of expressions are

```
31
OHBOY
X1*2-1
-29
AB1-AB2+5
```

EVALUATION OF ARITHMETIC EXPRESSIONS

An arithmetic expression is evaluated as follows. First, each element is replaced by its numerical value. Second, each term is evaluated by performing the indicated multiplications and divisions from left to right, in the order in which they occur. In division, the integral part of the quotient is retained, and the remainder is discarded. For example, the value of the term '5/2*2' is 4. In the evaluation of an expression (or any part of it), division by zero is regarded as an error. Third, the terms are combined from left to right in the order in which they occur. If the result is negative, it is replaced by its two's complement. Finally, the result is reduced MODULO 2^{35} (except in the variable field of a 'VFD' pseudo-op). All evaluations are done with full-word arithmetic (i. e., 36-bit signed arithmetic), only the final result is truncated to the proper length. An expression is undefined if any part of it is undefined, or if any error occurs in evaluating it. An undefined expression has the value zero.

INTEGER CONVERSION MODE

At the beginning of an assembly, UMAP assumes that all integers encountered are in decimal mode. This conversion mode may be altered for large segments of the program (a change in the global conversion mode), or for a single card or part of a card (a change in the local conversion mode). Local mode changes may be made through usage of a qualifier or by punching an '8' or '0' (zero) in column 7. (An '8' in column 7 is equivalent to a /K/ before the location field, while a '0' in column 7 is equivalent to a /D/ before the location field). Local changes are reset before the next card is processed. The global mode is always reset between cards to whatever it was before the last card. The global mode may be modified through usage of the 'SAK', 'OCTMOD', or 'DECMOD' pseudo-ops. When octal mode is in effect, the occurrence of a decimal integer will cause an expression to be treated as an undefined expression. The integer conversion mode in effect applies to integers in all fields of a card.

BOOLEAN EXPRESSIONS

An expression is Boolean if and only if

(1) It forms the variable field of a 'BOOL' pseudo-op, or

(2) It forms a Boolean subfield of a 'VFD' pseudo-op variable field, or

(3) It follows the Boolean qualifier (/B/), or

(4) It forms the variable field of a Type-D machine instruction. The Type-D machine instructions are 'SIL', 'SIR', 'RIL', 'RIR', 'IIL', 'IIR', 'LNT', 'RNT', 'LFT', and 'RFT', and the extended sense indicator instructions 'SIB', 'RIB', 'IIB', 'BNT', and 'BFT'. Note that for the Type-D instructions Boolean mode is automatically set for the variable field evaluation. Boolean expressions and symbols are further defined as left-Boolean or right-Booolean (See the 'BOOL' pseudo-op and the '/L/' qualifier).

In Boolean mode, all integers are assumed to be octal. Further, the operations '+', '-', '*', and '/' have different meanings in Boolean mode:

\+ Inclusive or

\- Exclusive or

* And

/ One's complement

The bit relations which hold for these Boolean operations are as follows.

0+0 = 0	0-0 = 0	0*0 = 0	/0 = 1
0+1 = 1	0-1 = 1	0*1 = 0	/1 = 0
1+0 = 1	1-0 = 1	1*0 = 0	
1+1 = 1	1-1 = 0	1*1 = 1	

Although '/' is usually an operation involving only one operand (a unary operator), by convention 'A/B' is taken to mean 'A*/B'. The table for '/' as a binary operator is as follows.

0/0 = 0

0/1 = 0

1/0 = 1

1/1 = 0

Note that due to the fact that the '/' may occur either as a unary operator or a binary operator while the '/L/' qualifier is legal in Boolean expressions, there is one special case which is indeterminate which may arise in Boolean expressions. This occurs when 'L' is defined as a symbol and occurs in a Boolean expression preceded by a unary '/' and followed by a binary '/' (e. g., 'A+/L/B'). This difficulty is easily circumvented by not using 'L' as a symbol or by writing the binary '/' as a unary '/' (e. g., 'A+/L*/B'). Whenever the indeterminate case arises, UMAP assumes the '/L/' portion of the expression represents the left-Boolean qualifier. Other conventions in Boolean fields are as follows.

+A = A+ = A
-A = A- = A
A = A = 0
A/ = A
\+ = 0
\- = 0
* = 0
/ = 777777777777_8

The above tables and conventions define the four Boolean operations for one-bit quantities. The operations are extended to 36-bit quantities by the rule that each bit-position is treated independently.

EVALUATION OF BOOLEAN EXPRESSIONS

A Boolean expression is evaluated as follows. First, all integers are taken as octal and must be less than 2^{36}. Second, the operations '*' and '/' are carried out from left to right, all quantities being regarded as having 36 bits. Third, the operations '+' and '-' are carried out from left to right, all quantities being regarded as having 36 bits. The right-most 18 bits are preserved. The left-most 18 bits are dropped except in the variable field of a 'VFD' pseudo-op. Any use of a relocatable or an erasable symbol in a Boolean expression constitutes a Boolean error ('B' flag.) Since only the '/L/' qualifier is legal in a Boolean expression, once Boolean mode is entered in a subfield it cannot be turned off. A Boolean expression is a left-Boolean expression if a '/L/' occurs anywhere in it, or if a left-Boolean symbol occurs anywhere in it. Otherwise, a Boolean expression is a right-Boolean expression.

MODES OF EXPRESSIONS

In addition to evaluating expressions, UMAP must also decide for each expression whether its mode is absolute, relocatable, or erasable. This is necessary in order to assign the proper relocation indicator bits for the information of the loader. The rule by which this decision is made is unavoidably complex, but fortunately expressions normally assume rather simple structures.

Before describing the general rule for determining the mode of an expression, a list of the more commonly used simple rules and the commonly made errors will be given. These should give some insight into the meaning of the general rule. The following simple rules may be stated.

(1) A relocatable element is a relocatable expression.

(2) A relocatable element plus or minus an absolute element is a relocatable expression.

(3) An absolute element is an absolute expression.

(4) Any expression containing only absolute elements is an absolute expression. (Thus, a Boolean expression has absolute mode.)

(5) The difference of two relocatable elements is an absolute expression.

(6) An erasable element is an erasable expression.

(7) An erasable element plus or minus an absolute element is an erasable expression.

(8) The difference of two erasable elements is an absolute expression.

The following errors are quite commonly made in writing UMAP expressions. All of these are relocation errors ('R' FLAG).

(1) The negative of a relocatable element.

(2) The negative of an erasable element.

(3) An absolute element minus a relocatable element.

(4) An absolute element minus an erasable element.

(5) The sum of two relocatable elements.

(6) The sum of two erasable elements.

(7) The sum of a relocatable element and an erasable element.

(8) The product of two relocatable elements.

(9) The product of two erasable elements.

(10) The product of a relocatable element and an erasable element.

A precise rule will now be given which applies to all expressions, however complicated. First, discard any term which contains only absolute elements. Then examine each remaining term of the expression. If any term contains more than one relocatable element, more than one erasable element, or one relocatable element and one erasable element, then the expression is a relocation error. Also, if in any term the character '/' follows the occurrence of a relocatable element or of an erasable element, then the expression is a relocation error. If the expression passes these tests, then do the following. Replace each relocatable element by an 'R', each erasable element by an 'E', and each absolute element by its value. This yields an expression in R and E with constant coefficients. Evaluate the expression as in normal algebra. Then, if the result is R, the expression is relocatable. If the result is E, the expression is erasable. If the result is numeric, the expression is absolute. Any other result indicates a relocation error.

Let R1 and R2 be relocatable symbols and N be an absolute symbol. Then consider the expression

N*R1-R2+3*N-2

Following the above procedure, eliminate all absolute terms, leaving

N*R1-R2

These terms contain no error. Then replace with R and get

N*R-R

If N has the value 1, this is an absolute expression, since 1*R-R = 0. If N has the value 2, this is a relocatable expression, since 2*R-R = R. For any other value of N, this is a relocation error.

The expressions

**

and

-

Are often used to indicate an address or a decrement computed by the program at execution time. Each of these is an absolute expression whose value is zero.

In an absolute assembly, all symbols, and hence all expressions, are absolute. A relocation error is impossible in absolute assemblies.

UMAP CARD FORMAT

The following symbolic card format is used for UMAP instructions--

CARD COLUMNS	FIELD NAME	TERMINATION
1-6	Location	Column 6
7	Print, Integer Mode Control	Column 7
8-14	Operation	Blank
		*
		'
		(
16-72	Variable	Blank Column 72
73-80	Card ID	Column 80

If the operation field is terminated by a blank, then the variable field starts in the first non-blank column thereafter but before Column 17. The same is true if the operation field is terminated by an '*'. If the operation field is terminated by a comma, then the variable field starts in the column immediately following the comma. Finally, if the operation field terminates with a left parenthesis, then the variable field begins with that left parenthesis. The operation field always begins in column 8. Hence, in light of the above conventions, it is possible for the variable field to begin as early as column 8 or column 9.

LOCATION FIELD (Columns 1 To 6).

When UMAP translates a program into the binary equivalent program, it assumes that storage locations are to be assigned consecutively starting at zero. Thus, the programmer need not worry about the actual assignment to storage of his program. However, there are many instances in which the program must refer to some part of itself - e. g., a transfer to some new instruction which is not in sequence, or a reference to some storage location reserved by the program. UMAP makes this reference easy by allowing use of 'symbolic' references within the program. If a particular instruction is to be referenced from some other part of the program, a symbol may be placed in its location field, and all references to the instruction may be made by means of this symbol.

For example, in the instruction

```
ALP    STO    BETA
```

'ALP' is the location field symbol. To reference this instruction elsewhere, the symbol 'ALP' may be used as follows:

```
TRA    ALP
```

The symbol 'ALP' is automatically defined, by its occurrence in a location field, as the equivalent of some machine location. Therefore, in the translation process, whenever 'ALP' occurs in an expression it is replaced by the numerical value which is the equivalent machine location. UMAP keeps a list (The 'Symbol Table') of all symbols and their values, so that a programmer may make all references in terms of symbols and is freed from the task of machine location assignment.

The location field of a machine instruction should be blank if the instruction is never referred to. The location fields of the various pseudo-ops will sometimes require symbols. This depends upon the particular pseudo-op and its function.

If column 1 of the card contains an '*', then the card is assumed to be a comment card. This card is printed out in the assembly listing but does not otherwise affect the program.

If an integer occurs in the location field, then the value of the integer (converted according to the prevailing integer conversion mode) is used to reset the storage location counter (see the 'ORG' pseudo-op) before the card is processed. This is normally used in override assemblies only, and usually causes phase errors if UMAP is producing the program card for the program.

A final convention which applies to location fields is that if a '+' or '-' occurs in one, then the field is treated as all blank. This allows the programmer to punch a count in the location fields of cards near one bearing a location field symbol. This may be used to indicate that references are made to the symbol plus or minus a constant. Thus, one might punch

```
ALPHA     CLA    =7
+1        STO    PAC
+2        ADD    ZIP
```

Where elsewhere in the program there occurs

```
TRA    ALPHA+2
```

PRINT AND INTEGER MODE CONTROL (Column 7)

Ordinarily, column 7 is blank on a UMAP card. This column may however be used to control the mode of integer conversion on the card, or to control the print mode of the card. UMAP recognizes the following four characters in column 7 -- '0', '8', 'P', and 'N'. Any other character in column 7 is ignored i. e., is treated as a blank. The interpretations of the above four characters (in column 7 only) is as follows.

1) '0' in column 7 -- This indicates that all integers on the card are to be treated as decimal integers. This is equivalent to a /D/ before the card is analyzed and may be reset by qualifiers appearing on the card. (See the sub-section on integer conversion mode.)

2) '8' in column 7 -- This indicates that all integers on the card are to be treated as octal integers. This is equivalent to a /K/ before the card is analyzed and may be reset by qualifiers appearing on the card. (See the sub-section on integer conversion mode.)

3) 'P' in column 7 -- This indicates that the card is to be printed in the assembly listing. 'P' in column 7 overrides for one statement the effect of such pseudo-operations as 'NOLIST', 'PCC', 'PMC', and 'BRIEF'. Thus, individual cards bearing some particular importance may be printed in full, while the remainder of the assembly is printed more briefly. The effect of a 'P' in column 7 carries over to a following 'ETC' card where such is legal and does occur.

4) 'N' in column 7 -- This indicates that the card is not to be printed in the assembly listing. This overrides for one statement any other print control. However, if there is an error on the card (fatal or non-fatal), it will be printed. Erroneous card printing cannot be overridden. The effect of 'N' in column 7 carries over to any following 'ETC' cards where such are legal and do exist.

OPERATION FIELD (Columns 8 to 14)

Machine instructions have a unique operation code which is recognized by UMAP. Whereas the symbols appearing in the location field are arbitrary, the operation codes are not. They must be those specified in this section. The sub-section on operation codes gives all the codes which UMAP will recognize in the operation field. (Note, however, that through use of the 'OPSYN' pseudo-op and MACRO definitions, the programmer may introduce new operation codes.)

Certain operation codes may use indirect addressing. The convention in UMAP is to indicate that indirect addressing is desired by appending an '*' to the end of the operation code. If an operation code has an '*' appended, but cannot be indirectly addressed, an 'I' flag results.

If an operation code occurs which is not recognized by UMAP, or if an operation code is not terminated properly, UMAP ignores the remainder of the card. The card is treated as a 'BSS 1' and an 'O' flag is given. (See, however, the description of the 'NONOP' pseudo-op.)

An operation field may be left blank if desired. UMAP treats a blank operation field the same as a 'PZE'.

VARIABLE FIELD (Column 16 to 72)

The variable field has a variety of uses depending upon the operation code involved. For certain pseudo-ops the variable field is a list of symbols (e. g., 'ASSIGN', 'EXTERN'), while for others it is a list of expressions (e.g., 'CALL', 'READ'), and for still others it is a list of data items (e. g., 'DEC', 'OCT', 'VFD'). The form and meaning of the variable field for pseudo-ops is described for each pseudo-op later in this section.

For machine instructions, the variable field is usually divided into three subfields; The address subfield, The tag subfield, and The decrement subfield. In the 7090/7094 a word is 36 bits, and the four parts of this word are as follows.

PREFIX	BITS	1-3
DECREMENT	BITS	4-18
TAG	BITS	19-21
ADDRESS	BITS	22-36

For certain machine instructions, an address, decrement, and tag are required, while for others only an address is required (with a tag optional), and for still others no variable field should be given at all. The programmer must know, for each machine instruction, what information is required.

The various subfields of the variable field may be given in symbolic form, with UMAP translating from the symbolic to the binary form. The address, tag, and decrement subfields may all be given as expressions.

The variable field starts, on a card, after the termination of the operation field. In all cases it must begin no later than column 16. A blank variable field is equivalent to a zero variable field. The same applies to any subfield of the variable field. The variable field is terminated by the first blank after the beginning of the variable field. Any comments punched on a card after the variable field has terminated are printed with the card but do not affect the processing of the card. If column 72 is encountered, the variable field is forced to end. Columns 73 through 80 are never processed by UMAP.

The three subfields are written in the order 'Address Subfield', 'Tag Subfield', 'Decrement Subfield' on a card, with the subfields separated by commas. (Note that this order is the opposite of that in which these fields occur in the resultant binary word.) It is permissible to leave blank any of these subfields which are not needed on a card, but only from right to left, since the first blank terminates the card. Thus, the following forms may occur in the variable field.

```
OP      ADD
OP      ,TAG
OP      ,,DEC
OP      ADD,TAG
OP      ADD,,DEC
OP      ,TAG,DEC
OP      ADD,TAG,DEC
```

Note that commas (or adjacent commas) may be used to delete a field. Thus,

```
OP    ,,DEC
```

Is the same as

```
OP    0,,DEC
```

Or

```
OP    0,0,DEC
```

The address subfield usually specifies a machine location, in which case it must be relocatable if the assembly is relocatable. For some machine instructions, however, the address requires an absolute value which is not a machine location (e. g., the shift instructions.) Thus, the following sequence might occur.

```
ZAC
LDQ   A+5
LGL   6
SLW   BIB1
SUB   BLANK
TZE   K3
```

The tag subfield is used to refer to index registers. Usually the actual integer is given as the tag subfield, but an absolute expression may be given if desired. Thus, an example sequence of coding is:

```
AXT   10,2
CLA   A+10,2
STO   ARG
TSX   SUB,4
PAR   ARG
STO   B+10,2
```

A tag subfield is computed modulo 8. If more than 3 bits are generated by the instruction for the tag field, a 'T' flag results.

The decrement subfield most often is used with a set of instructions which deal with the index registers (e. g., 'TXL', 'TIX', 'TNX'). In such cases the decrement contains a number by which an index register is to be changed or against which it is to be compared. Usually such a number is absolute, but in some cases a relocatable decrement may be desired. Again, the decrement may be an expression. An example sequence is:

```
LXA   N,4
CLA   A1,4
FAD   C2A+10,4
XCA
FMP   I-J+KM1,4
STO   A1,4
TIX   *-5,4,2
```

CARD ID FIELD (Columns 73 to 80)

Columns 73 to 80 of a card are always ignored by UMAP, and hence these columns may have any desired BCD punching in them. For the programmer's convenience (and safety) these columns may be used to provide a sequenced identification for the UMAP deck. Any punching in columns 73 to 80 is printed in the assembly listing, along with the card, and thus aids in relating segments of the assembly listing and the source deck.

QUALIFIERS

'Qualifiers' are entities which allow local modifications to be made in the scanning and interpreting of variable field expressions. The effect of most qualifiers is limited to the duration of the card being scanned, and to at least the next element in the variable field. The form of a qualifier is '/S/', where S is a symbol designating the desired qualifier. UMAP recognizes qualifiers only if the initial slash (/) initiates the variable field or immediately follows a break character. A qualifier appearing in any other context will result in an error during the scanning of the expression. UMAP recognizes fourteen qualifiers, seven of which are associated with MACROS and will not be discussed here (See the MACRO sub-section.) The seven remaining qualifiers are as follows.

/B/ - BOOLEAN QUALIFIER
The remainder of the present subfield is evaluated in Boolean mode. Boolean mode is turned off at the end of a subfield. For proper functioning, a '/B/' should be the first thing in a subfield.

/D/ - DECIMAL QUALIFIER
All integers remaining in the variable field are evaluated in decimal mode.

/H/ - HOLLERITH QUALIFIER
The BCD representation of the following symbol (right-justified with leading zeroes) is used in the evaluation of the expression - as opposed to the 'defined value' of the symbol.

/K/ - OCTAL QUALIFIER
All integers remaining in the variable field are evaluated in octal mode. This qualifier may also be written as /O/.

/L/ - LEFT-BOOLEAN QUALIFIER
Appearance of '/L/' anywhere in a Boolean expression causes the expression to be defined as a left-Boolean expression. '/L/' is a legal qualifier only if it occurs in a Boolean expression.

/R/ - RELOCATION QUALIFIER
Causes the following integer to be computed modulo 2^{15} and treated as a relocatable address. This qualifier may be used in relocatable assemblies only.

QUALIFIERS (Continued)

/TV/ - TRANSFER VECTOR QUALIFIER
Causes the following symbol to be defined by entering it into the transfer vector. This qualifier is legal only if UMAP is generating the program card.

Qualifiers may never occur in the location or operation subfields of a card. If an undefined or illegal qualifier occurs in a subfield, the subfield is treated as undefined and a 'Q' flag is given.

LITERALS

The constant-generating pseudo-ops (Such as 'OCT', 'DEC', 'VFD', 'BCI') are available for the generation of tables of constants, but are often inconvenient for the inclusion of single constants. To facilitate the usage of individual constants, an entity, known as a 'Literal', is recognized by UMAP.

In contrast to other types of subfields, a literal subfield contains the actual data to be operated upon. Thus, references to a particular constant may be done by simply giving the constant itself plus a key character which indicates that the constant is a literal. The appearance of a literal directs UMAP to translate it as would be done if it occurred in a constant-generating pseudo-op, store the resultant constant at the end of the program (along with the other literals), and replace the literal subfield with the location in which the literal is stored.

A literal is formed by preceding the desired constant with an equal sign (=), possibly followed by a second character. Three literals are recognized by UMAP. These are:

=H - HOLLERITH LITERAL
The next six characters in the variable field (including break characters) after the 'H' are taken as a BCD constant. For example,

```
CAS     =HABC123
```

=HABC123 converts into a constant whose octal equivalent is 212223010203.

=K - OCTAL LITERAL
The number following the 'K' is converted as an octal integer. This number may have one of two forms -- 'N' or 'NKE', where N is an octal integer of 12 or fewer digits, with or without a sign, and E is a predefined, absolute expression. Let M = value of E. Then, N is shifted M octal places to the left, if M is greater than zero, or M octal places to the right, if M is negative. For example,

```
CAL     =K71326
LDQ     =K71K4
```

=K71K4 Generates the constant 000000710000. The octal literal may also be written as =0.

= - DECIMAL LITERAL
The number following the equal sign (=) is converted as a decimal number. The conventions concerning 'B' and 'E', floating point and integer mode, apply here as on the 'DEC' card. For example,

```
FMP     =0.1E1
```

=0.1E1 Generates the constant whose octal equivalence is 201400000000.

LITERALS (Continued)

Literals may occur in almost any subfield that requires a machine location. Thus, literals may occur in the address and decrement subfields of machine instructions (except type D instructions), in the variable fields of 'CALL' or 'CALLIO' type statements, on 'ETC' cards when they are used to extend 'CALL' or 'CALLIO' type statements and on 'RELIST', 'EQU', 'SYN', and 'SET' cards. Literals cannot occur if Boolean mode is in force, and they cannot occur in a 'VFD'. Whenever a literal occurs, it must be the entire subfield. Literals cannot form part of an expression. Other subfields may follow a literal subfield, if desired.

The constants generated by literals are converted to binary in PASS 1 and saved in a table. A particular binary literal occurs only once in this table, no matter how many times it occurs symbolicly in the program. Thus, the literals =H00000A, =K21, and =17 would generate the same binary literal and only one entry to the literal table. Between passes, this binary table is sorted according to 36-bit logical comparisons and assigned to successively higher locations following the highest location used by the program itself. An example sequence using a literal is:

```
        ZAC
        LDQ     A+5
        LGL     6
        SLW     BIB1
        SUB     =H00000
        TZE     K3
```

This sequence could also be written using

```
        SUB     =K60
```

Another sequence using literals is,

```
        AXT     60,1
ALP     CLA     RAD+60,1
        FDP     =2.0
        FMP     =3.14159
        STO     P2RAD+60,1
        TIX     ALP,1,1
```

In a relocatable program, a literal is a relocatable expression, while in an absolute program, a literal is an absolute expression. If a literal occurs illegally in a subfield, or if a literal is improperly formed, the subfield is treated as an undefined subfield and a 'L' flag is given.

Usually, the literals are printed at the end of an assembly following the 'END' card and any 'RMT' assembly sequences. This may be changed by use or the 'PUNLIT' pseudo-op or by use of the 'BRIEF' pseudo-op, which controls the printing of the program literals. Neither of these pseudo-ops affects the storage assignment of the literals.

CALLING SEQUENCES IN UMAP

A 'calling sequence' is a set of machine words used to call into action a subroutine and to specify the parameters needed by the subroutine for its execution. Usually a calling sequence begins with a 'TSX' instruction transferring to the subroutine (possibly through a transfer vector) followed by those words necessary for specifying the parameters. If the subroutine is 'external' to the calling program, then the transfer to it must pass through a transfer vector. This requirement results from the method of loading of relocatable subroutines. All subroutines in the MAMOS library must be entered via a 'TSX' on index register four. Subroutines written by the programmer may use any technique desired to establish contact between a calling program and a called program, but external subroutines should be entered through a transfer vector. Internal subroutines may use any method of entry. No transfer vector is used for these.

The structure of a calling sequence obviously depends upon the subroutine called. A subroutine may require any desired structure. Subroutines written by a programmer may thus have any arbitrary calling sequence. Subroutines in the MAMOS system fall into two classes with respect to calling sequences -- I/O (Input/Output) type calling sequence and non-I/O type calling sequence. To provide for these two types of calling sequences, two pseudo-ops are provided in UMAP -- 'CALL' and 'CALLIO'. 'CALL' is used to set up a non-I/O type calling sequence. 'CALL' and 'CALLIO' provide two functions. First, they enter a subroutine name into the transfer vector, thus defining this symbol. This occurs only if UMAP is generating the program card, and in this case, the symbol in the first subfield of the variable field is entered into the transfer vector. If UMAP is not generating the transfer vector, then 'CALL' and 'CALLIO' may be used, but the program itself must define the subroutine name. Second, the remainder of the variable field of the 'CALL' or 'CALLIO' contains the calling sequence. Two types of parameters may occur here, the 'SINGLE' parameter or the 'BLOCK' parameter. The single parameter is set apart by commas, while the notation ',...,' is used to indicate block parameters. Thus,

```
CALLIO  .PRINT,F,A,B,...,C,D,0
```

Is an I/O call for the '.PRINT' subroutine, with a format named 'F', to read 'A', 'B' to 'C' inclusive, and 'D'. (I/O calling sequences always end with a 'zero parameter', i. e., a word with zero in its address, tag, and decrement.) This calling sequence assembles the same as

```
CALLIO  .PRINT
FMT     F
IOP     A
IOP     B,,C
IOP     D
ENDIO
```

Where 'FMT', 'IOP', and 'ENDIO' are extended machine instructions used for I/O type calls. An example of a non-I/O type call is,

```
CALL    ZERO,X,Y,...,Y+20,Z
```

Which is the same as

```
CALL    ZERO
PAR     X
BLK     Y,,Y+20
PAR     Z
```

Where 'PAR' and 'BLK' are extended machine instructions. 'PAR' is used for single parameters and 'BLK' for block designations. The first instruction of the last calling sequence could also be written as follows.

```
TSX     /TV/ZERO,4
```

A number of special pseudo-ops are built into UMAP to provide ease in calling certain subroutines available in the system library,

PSEUDO-OP	ASSEMBLES AS
COMMNT	CALLIO .CMMNT
LOOK	CALLIO .LOOK
PAUSE	CALL .PAUSE
PRINT	CALLIO .PRINT
PUNCH	CALLIO .PUNCH
READ	CALLIO .READ
RESTOR	CALLIO .RSTOR
SAVE	CALLIO .SAVE
SETTO	CALLIO .SET
TAPERD	CALLIO .TAPRD
TAPEWR	CALLIO .TAPWR

The variable fields for all of these pseudo-ops are the same as for 'CALLIO', except that the first element of the variable field is not the name of a subroutine since this is specified by the particular pseudo-op. Thus, the statement,

```
READ    =H80C1* ,CARD,...,CARD+79,0
```

Is equivalent to the statement

```
CALLIO  .READ,=H80C1* ,CARD,...,CARD+79,0
```

And could also be written

```
READ
FMT     =H80C1*
IOP     CARD,,CARD+79
ENDIO
```

PROGRAM CARDS

Normally, UMAP produces atuomatically a program card and a transfer vector for a program. In order that this automatic feature function correctly, the following rules must be observed.

(1) All program common storage must be defined by use of the 'PCLIST' and 'PGMCOM' pseudo-ops.

(2) All erasable storage must be defined by use of the 'ERLIST' and 'ERAS' pseudo-ops.

(3) All external subroutines used by the program must be called by use of the 'CALL' or 'CALLIO' type pseudo-ops or by use of the /TV/ qualifier. Internal subroutines cannot be called by use of the 'CALL' or 'CALLIO' pseudo-ops or by use of the /TV/. This will result in multiply-defined symbols.

(4) If the program being assembled is a subroutine, then all the entry points must be named in the variable fields of one or more 'ENTRY' pseudo-ops.

Violation of any of these rules can result in programs which will not load for execution or which load and/or execute incorrectly.

There are those cases in which an automatic program card is not desired, e. g., If a non-standard program card is needed or if the assembly is a symbolic override assembly. It is possible to delete the automatic program card. The occurrence in the program of any one of the following pseudo-ops will do so, 'ABS', 'ENDPGM', 'FULL', 'PGM', 'REL'. If any one of these pseudo-ops occurs, then the program itself is fully responsible for generating both the program card (if one is desired) and the transfer vector (if one is needed). Further, the /TV/ qualifier and several pseudo-ops are not recognized by UMAP if the program card is off, and other pseudo-ops (e. g., 'CALL' and 'CALLIO') are processed somewhat differently. The pseudo-ops so affected are noted later.

Pseudo-ops are available for facilitating the manual production of program cards. Two pseudo-ops especially, 'PGM' and 'ENDPGM', are necessary for this purpose. The 'PGM' informs UMAP that the cards which follow are to be assembled and punched as a program card. This is necessary since program cards have a different format than normal relocatable binary cards. The 'PGM' is then followed by the symbolic cards which represent the desired program card. It is assumed here that the reader knows the format and information content of a program card. The 'ENDPGM' pseudo-op simply informs UMAP that the symbolic program card has ended. UMAP punches the program card, resets the punch mode to relocatable binary, and sets the assembly mode to relocatable, zeroing the storage location counter. This must, of course, be immediately followed by program common storage assignment (if there is any) and the transfer vector.

As an example, consider the following subroutine named 'MINCOS', which has three arguments (x,y,z), and which sets Z equal to the smaller of COS X and SIN Y. The calling sequence for MINCOS is,

```
        CALL    MINCOS,X,Y,Z
```

The subroutine could be written,

```
        PGM                     PROGRAM CARD
        PZE     G,,2            LENGTH,,NO. OF SUBS
        PZE     -1              ERASABLE,,PROGRAM COMMON
        BCI     1,MINCOS        PROGRAM NAME
        PZE     B               ENTRY POINT
        ENDPGM
COS     BCI     1,COS           TRANSFER VECTOR
SUB     BCI     1,SIN
B       CLA     1,4             START SUBROUTINE
        STA     C               SET X
        CLA     2,4
        STA     D               SET Y
        SXA     A,4
        CALL    COS             COMPUTE COS X
C       PAR     **
        STO     F               SAVE COS X
        CALL    SUB             COMPUTE SIN Y
D       PAR     **
        CAS     F               COMPARE AND PUT
        CLA     F               SMALLER IN AC
        NOP
A       AXT     **,4
        STO*    3,4             STORE IN Z
        TRA     4,4
F       ERAS
G       END
```

Note that in the above program the subroutine 'SIN' is called by the name 'SUB' within the program itself. This can be done when the program is constructing its own program card, but it is otherwise impossible.

SYMBOLIC OVERRIDES TO BINARY PROGRAMS

At times it is desirable to make small changes in a binary deck, as opposed to re-translating the entire symbolic source deck, to make corrections in a program. Since the MAMOS system allows the mixing of binary and symbolic segments in a given job, such changes may be made with UMAP assemblies instead of manually punching binary cards. Such changes are called 'OVERRIDES' and may be made to any binary deck, regardless of the original source language.

The assembly mode of an override should agree with the translation mode of the original deck. (MAD and ALGOL compilations are always of relocatable mode, while UMAP assemblies may be absolute or relocatable.) The override section should follow, physically, those instructions in the deck which are being overridden. Note that what happens is that the original incorrect words are loaded with the program, but are then replaced by the override words when they are loaded.

Certain handy conventions are available in UMAP for writing overrides. A numeric location field is treated as an origin before the card is processed, so that 'ORG' cards are generally not needed in overrides. Further, since in overrides one usually refers to octal locations, the 'SAK' or 'OCTMOD' pseudo-ops may be used to set octal mode conversion for all integers. This may be changed locally, for one card, by use of the '0' or '8' punched in column 7 of the card. (See integer conversion mode sub-section.)

Since UMAP produces a program card unless told to do otherwise, the programmer must, in an override, turn this feature off. This is done through use of the 'ABS' or 'REL' pseudo-ops, depending upon the assembly mode desired. Thus, for example, the following absolute override,

```
        ABS
        OCTMOD
17320   CLA     21356
16142   STO     17300
        STO     17301
        END
```

Causes a 'CLA 21356_8' to be loaded at 17320_8, a 'STO 17300_8' to be loaded at 16142_8, and a 'STO 17301_8' to be loaded at 16143_8. In any override, the origin, for each correction, must correspond to the location of the instructions being corrected, but corrections to consecutive core locations need have an origin for the first of the sequence only. When in octal mode, care must be taken that all integers are written in octal. For example, if one writes,

```
        LGL     12
```

With octal mode in effect, it is equivalent to

```
        LGL     10
```

In decimal mode. If a decimal 12 is desired, it should be written

```
        OLGL    12
```

Or

```
        LGL     /D/12
```

Or

```
        LGL     14
```

Octal mode will still be in effect for the next card.

Any address or decrement subfield which is numeric is assumed to be absolute. In absolute overrides this is desired, but in relocatable overrides one usually desires relocatable addresses and sometimes decrements. The /R/ qualifier is used to obtain such relocatable subfields.

Thus, an override might appear as,

```
        REL
        SAK
1760    FMP     /R/1702,2
        STO     /R/371,1
        END
```

Here, the 'FMP' and the 'STO' will have relocatable addresses when loaded. If in an override a particular address occurs often, it is easier to define a relocatable symbol with an 'EQU' card and use the symbol in the override. An example would be,

```
        REL
        SAK
A       EQU     /R/106
1130    CLA     /R/71
        STO     A
1136    STO     A
1561    CLA     A
        STO     /R/107
        END
```

Certain difficulties arise with relocatable overrides which do not occur with absolute overrides. First, due to the manner of loading a relocatable program, a relocatable override cannot be put just anywhere later in the deck. An override must be in the same program segment as the instruction it overloads, i. e., there must be no program cards between them. This is due to the fact that the relocation constant changes every time a program card is encountered.

A second problem occurs in the assignment of relocation bits to symbols. Relocatable symbols are handled in one of two ways at loading time, depending upon whether the symbol refers to a location inside the present program or outside of it. Normally the user doesn't need to worry about this, since UMAP is handling the relocation assignment and has available the entire program for its information. For overrides, however, UMAP does not have at its disposal the information concerning the original program, so the user must supply part of it. For example, consider the following override.

```
        REL
        OCTMOD
123     CLA     /R/163,4
        STO     /R/100,4
        FAD     /R/63,4
        END
```

This override is assembled as a separate program whose length is 126_8. The address of the 'CLA' is 163_8, and hence is outside of this program, while the addresses of the 'STO' and the 'FAD' are inside the program. If the original program is longer than 163_8, this override is incorrect, since in this case the 163_8, should assemble as being inside the program. Thus, suppose the original program has a length of 173_8. To make the override correct, the following convention may be used; If the 'END' card has a numeric location field, then this number is taken as the program length.

Thus, the above override should end with

```
        173     END
```

Note that the previous relocatable overrides will all assemble correctly, since each override actually appears longer, to UMAP, than any address in it. The general rule for relocatable overrides is to punch, in the location field of the 'END' card, the length of the program being overridden.

At times an override requires the addition of instructions to a program as well as the changing of instructions already in it. In this case 'patches' must be made. Here a set of instructions are loaded into some area not used by the program and 'TRA' instructions into this area are loaded onto the area to be corrected. Such a patch area must be within the given program, for relocation purposes, but not used. Often programs specifically set aside an area for patches. In overriding with patches it is possible that information on the program card is no longer correct, so that a new program card must be prepared. Through use of the 'BINARY' pseudo-op, the program card and override can be assembled in one assembly. However, it is recommended that in such cases the corrections be made in the symbolic source deck with a re-translation, rather than with overrides.

Finally, literals should never occur in an override of any kind. Also, program common assignment cannot occur, and no new erasable assignment should occur. However, since erasable symbols are handled as special relocatable symbols, overrides should refer to erasable locations by defining erasable symbols and using them for all references.

ERROR COMMENTS

Normally, UMAP is able to complete two passes over the program despite any errors which may be found in the program. However, certain types of errors may occur which prevent the continuation of the assembly process, and thus the listing is either incomplete or not given at all. For such cases, UMAP prints out error comments describing the trouble. Also, the card being analyzed at the time of the trouble is printed. The possible error comments are as follows,

COMMENT	TROUBLE
Literal Table exceeded	More than 200 different binary literals.
Entry Table exceeded	More than 50 different entry points.
Operation Table overflow	More than 2000 new operations defined.
Transfer Vector overflow	More than 50 different subroutines called.
Symbol Table overflow	Too many symbols in program - See the 'SYMBOL' pseudo-operation.
Macro Table exceeded	Too many macro definitions or too many remote assembly sequences.
Compile Table exceeded	Macro calls nested too deeply.
Symbol Table check-sum	Bad check-sum in symbol table deck.
Missing 'END' card	No 'END' card.
Created symbols exceeded	More than 1000 created symbols.
'ETC' generated in 'RMT'	Macro expansion caused an 'ETC' card to be generated in a 'RMT' sequence.

In addition, there are those times when UMAP may suspect a machine error. In such cases, assembly is halted with the comment,

'Possible machine error here - assembly discontinued'

ERROR FLAGS

In the left-hand margin of an assembly listing produced by UMAP there will sometimes occur 'ERROR FLAGS'. These flags are single letters, and their presence on a line indicates that UMAP has found what appears to be an error on the card printed on this line. It is possible for several flags to occur on a given line. A complete list of these flags, with a description of their associated error, follows.

FLAG	ERROR
A	Address missing, or address given but normally not used.
B	Error in Boolean expression.
D	Decrement missing, or decrement given but normally not used.
E	Missing or illegal 'ETC' card.
G	Error in generation of program constants.
I	Indirect addressing illegally specified.
L	Incorrect literal construction, or a literal occurs illegally.
M	Multiply-defined symbol on card.
METC	Missing 'ETC' card in a macro call.
N	Non-fatal error on card.
O	Undefined operation code on card.
P	Phase error - expression should be pre-defined, but isn't, or a symbol's definition, given on pass 1, does not check on pass 2.
Q	Error in qualifier specification, or qualifier illegally used.
R	Error in formation of relocatable expression.
T	Tag missing, or tag given but normally not used.
U	Undefined symbol on card.
UPAR	Unmatched parentheses in macro call.

The flags 'A', 'D', 'I', 'N', 'T', and those obtained by the 'FLAGOP' or 'FLAGSY' pseudo-ops are considered as non-fatal flags. All other flags are considered as fatal. The occurrence of one or more fatal flags in an assembly causes the assembly to be unsuccessful, and execution is not allowed.

UNDEFINED AND MULTIPLY-DEFINED SYMBOLS

Undefined symbols ('U' flag) are symbols which are used in the program but which are never defined by the program. Multiply-defined symbols ('M' flag) are symbols which are given two or more different definitions. A symbol may be defined as often as desired if all definitions agree. Only when the definitions disagree is the symbol multiply-defined. Near the end of the listing, a list of all undefined symbols is printed, followed by a list of all multiply-defined symbols. Also, every occurrence of such symbols is appropriately flagged.

REFERENCE TABLES

At the end of the listing two reference tables are printed. The first is the symbol reference table. Each symbol defined is listed, with its mode, its value, and the locations of all references in the program to this symbol. The symbol mode is indicated by one of four letters:

A	ABSOLUTE (Includes program common)
E	ERASABLE
R	RELOCATABLE
T	TRANSFER VECTOR

The second reference table is for the literals. All references to each literal are printed out. There are pseudo-ops in UMAP which allow the collection of references to symbols and to literals to be turned off. See the pseudo-op subsection.

UMAP OPERATION CODES

In the following descriptions, a pre-defined expression is an arithmetic expression all of whose components are 'defined' at the time the expression is encountered on PASS 1 processing by UMAP. (I. e., Each symbol in the expression must already be in the symbol table.) Throughout the assembly process, a location counter 'L' is kept by UMAP to count the number of locations used by the program. This counter is updated for each location used by the program, and it is used to define many of the symbols occurring in the program. Always, L is set to the next available location for the program. L is initially zero. Not all operation codes accepted by UMAP are described in the following. See the MACRO sub-section for the descriptions of all macro-related operation codes.

PSEUDO-OPERATIONS

9LP - 9 LEFT PREFIX

Causes subsequent binary cards (until the next 'FUL', 'ABS', 'REL', or 'PGM' pseudo-op occurrence) to be punched in absolute mode with a prefix on the first word of the card of N, where N is the value of the absolute expression in the variable field of the '9LP'. Deletes the automatic program card feature. N is taken modulo 3.

ABS - ABSOLUTE ASSEMBLY MODE

This pseudo-op causes the assembly to be in absolute mode deleting the automatic program card.

ASSIGN - ASSIGN STORAGE

The variable field is a list of symbols separated by commas. Expressions are not allowed. The symbols are assigned, in order of their appearance, to locations L, L+1, L+2, etc. Note that these symbols are assigned at the point at which the 'ASSIGN' is found. Adjacent commas may be used to obtain blocks.

Thus, for example, the statement

```
ASSIGN  A,,,,B,C,,,
```

Is equivalent to the statements

```
A    BSS    1
B    BTS    4
C    BSS    4
```

BCD - BINARY-CODED-DECIMAL

Causes the generation of 'N' words of BCD information, where 'N' is the integer in column 12 of the 'BCD' card. The words are stored in L through L+N-1. If column 12 is blank, 10 BCD words are taken from the card. In all cases, the first BCD word is assumed to start in column 13. An '*' in column 12 signals UMAP to compute its own word count. See the 'BCI' pseudo-operation. Location field symbol is defined as L.

BCI - BINARY-CODED-DECIMAL

The variable field consists of a one-digit integer count 'N', followed by a comma, followed by a string of Hollerith characters. The 6*N Hollerith characters after the comma are divided into groups of 6 characters (left to right) and stored at L, L+1, L+2,...., L+N-1. If ten words are desired, the variable field may start with a comma. In this case, the last word will be left-justified with trailing blanks. Blanks are included in a string of Hollerith characters. A convention is available with 'BCI' which deletes the need for the word count 'N'. One may write

```
BCI     *,Hollerith String
```

In this case UMAP counts the number of BCD words on the card and replaces the * by this count. This computed word count is obtained by deleting all blank words from the right-hand end of the variable field. A partial word on the right is left-justified with trailing blanks. Location field symbol is defined as L.

BES - BLOCK ENDED BY SYMBOL

N locations are reserved, where N is the value of the absolute, predefined expression in the variable field. A symbol in the location field is defined as L+N, i. e., as the first location after the block reserved.

BINARY - INSERT BINARY CARDS

This pseudo-op precedes a set of one or more binary cards which are stored during PASS 1 of UMAP and reinserted, during PASS 2, into the binary version of the program. There may be as many binary sections in a program as desired. If, in a binary segment, a symbol table is found, then the symbol table is read as in the 'RST' pseudo-op. Ordinarily, these binary segments are placed only on the execution tape in PASS 2.

However, if the symbol 'PUNCH' is found in columns 1-6 of the 'BINARY' card then the binary deck following it is placed on the output tape with new sequenced ID, corresponding to the ID being used with the binary object deck. Use of the 'BINARY' pseudo-op allows the assembly of several sections of overrides, interspersed throughout a binary deck, with only one call for UMAP and with a continuity of the symbol table between all symbolic sections.

BOOL - BOOLEAN EQUIVALENCE

The symbol in the location field is assigned the value of the expression in the variable field. In evaluating the variable field, Boolean mode is assumed. Eighteen bits are used in this definition, as opposed to the usual fifteen bit values given to symbols. (See the sub-section on PASS 2 symbol definition.) Symbols defined by 'BOOL' cards have absolute mode. The symbol is defined as a left-Boolean or a right-Boolean symbol according as the variable field is a left-Boolean or a right-Boolean expression. A Boolean expression is left-Boolean if any symbol in it is left-Boolean or if a '/L/' occurs anywhere in it; otherwise, the expression is a right-Boolean expression.

BRIEF - LISTING ABBREVIATION

After the occurrence of 'BRIEF' those pseudo-ops which usually produce several lines on a listing (e. g., 'DEC', 'OCT', 'VFD', 'CALL') are abbreviated by deleting all lines after the first one. Thus, the first object word generated plus the card itself are printed, but the remainder of the object words generated by this card are not printed. Further, the 'BRIEF' mode prevents the printing of the program literals.

BSS - BLOCK STARTED BY SYMBOL

N locations are reserved, where N is the value of the absolute, predefined expression in the variable field. A symbol in the location field is defined as L, i. e., as the first location in the reserved block.

BTS - BLOCK TERMINATED BY SYMBOL

N locations are reserved, where N is the value of the absolute, predefined expression in the variable field. A symbol in the location field is defined as L+N-1, i. e., as the last location in the reserved block.

CALL - NON-I/O SUBROUTINE CALL

The 'CALL' pseudo-op is used to call external subroutines or, if the automatic program card is deleted, internal subroutines. The variable field is a set of expressions separated by ',...,' or by commas. The first element of the variable field must be the name of the subroutine being called. If UMAP is generating the program card, this name is automatically defined and placed in the transfer vector. If the automatic program card is deleted, the program itself must define this symbol. The elements of the variable field following the subroutine name are assembled into a calling sequence. Single variables are separated by commas.

CALL - NON-I/O SUBROUTINE CALL - cont'd

Block parameters are indicated by the format A,...,B which means 'A to B, inclusive'. List elements may be symbols, constants, expressions, or literals. 'CALL' is used to call all ON-I/O type subroutines. A 'CALL' statement may be followed by one or more 'ETC' cards; in such cases, a subfield must end on the same card that it began on. A location field symbol is defined as the first location generated.

CALLIO - I/O SUBROUTINE CALL

This pseudo-op is the same as the 'CALL' pseudo-op, except that it is used to call I/O subroutines only. All comments concerning the variable field of a 'CALL' statement apply.

COMMNT - .COMNT CALL

This pseudo-op assembles the same as 'CALLIO .COMNT'. The variable field, except for the first element, is the same as for 'CALLIO'.

DATE - BCD DATE

This pseudo-op assembles, as two BCD words, the current date of the assembly. The first word has the form 'DDBMMM' and the second has the form 'BYYYYB', where B is blank, D is day, M is month, and Y is year. If there is a non-blank character in the variable field, this character replaces the final blank in the second word. A location field symbol is defined as the location of the first of the two words.

DEC - DECIMAL DATA

The decimal-data items in the variable field are converted to binary numbers and assigned to consecutive locations L, L+1, L+2, etc. Successive items in the variable field are separated by a comma. The first blank to the right in the variable field indicates that the field is terminated and that all remaining punching is a comment. A symbol in the location field is defined as the first location used by this card, i. e., as L. A 'DEC' card may not be followed by an 'ETC' card, but it can be followed by more 'DEC' cards. The obvious purpose of this pseudo-op is to introduce into the binary program sets of constants.

When the variable field is evaluated, adjacent commas cause the number zero to be generated, as does a comma followed by a blank. Thus, the number of words generated by a 'DEC' card is always one more than the number of commas in the variable field. If the variable field of a 'DEC' card contains anything other than a valid decimal-data item, the assembler will flag an error in the listing (G flag).

In the UMAP language, a decimal-data item is used to specify in decimal form a word of data to be converted to binary form and stored with the program. A decimal-data item may occur in one of two places in a UMAP program - in a decimal literal (See the literal sub-section) or on a 'DEC' card. Three types of decimal-data items are recognized by UMAP.

1) DECIMAL INTEGER

A decimal integer is composed of a string of digits possibly preceded by a plus or a minus sign. (Note - in all the following, a minus sign is required to indicate a negative integer, while a plus sign is not required for positive integers.) A decimal integer is distinguished from other types of decimal-data items by the fact that the letter 'B', the letter 'E', and the decimal point '.' are all absent.

2) FLOATING POINT NUMBER

A floating point number has two components as follows:

A) The principal part, which is a decimal number written with or without a decimal point. The decimal point may appear at the beginning or end of the principal part, or within the principal part, or may be omitted if the exponent part is present. If the decimal point is omitted, it is assumed to be located at the right-hand end of the principal part.

B) The exponent part, which consists of the letter 'E' followed by a signed or unsigned decimal integer. The exponent part must follow the principal part. It may be omitted if the principal part contains a decimal point.

A floating point number is distinguished from a decimal integer by the fact that either a decimal point or the letter 'E' (or both) must be present. It is distinguished from a fixed point number by the fact that the letter 'B' is absent.

3) FIXED POINT NUMBER

A fixed point number has three components as follows:

A) The principal part, which is a decimal number written with or without a decimal point. The decimal point may appear at the beginning or end of the principal part, or within the principal part, or may be omitted completely. If the decimal point is omitted, it is assumed to be located at the right-hand end of the principal part.

B) The exponent part, which consists of the letter 'E' followed by a signed or unsigned decimal integer. The exponent part may be absent. If present, it must follow the principal part, and may precede or follow the binary place part.

C) The binary place part, which consists of the letter 'B' followed by a signed or unsigned decimal integer. The binary place part must be present in a fixed point number, and must follow the principal part, but may either follow or precede the exponent part if there is one.

A fixed point number is distinguished from the other types of decimal-data items by the presence of the letter 'B'.

A decimal integer may represent any positive or negative binary number whose magnitude is less than 2^{35}. For example, the decimal integer '-31' would be converted to the 36-bit number whose octal representation is -000000000037, which is the same as 400000000037.

A floating point number will be converted to a normalized floating point binary word in the standard 7090 floating point binary format (See the 7090 manual). The exponent part, if present, specifies a power of 10 by which the principal part will be multiplied during conversion. For example, all of the following floating point numbers are equivalent and will be converted to the same floating point binary number.

```
3.14159
31.4159E-1
314159.E-5
314159E-5
.314159E1
```

A fixed point number is converted to a fixed point binary number which contains an 'understood' binary point. Note that in the 7090 numbers are either floating point or integral to the machine itself. The purpose of the binary place part of the number is to specify the location of this understood binary point within the 7090 word generated. Thus, the conversion of a fixed point number is done in the following steps:

1) The principal part, along with the exponent part, is converted to a binary number with a binary point. This is the usual decimal to binary conversion. Thus, the number 65B4E-1 would first be converted to the binary number 110.1, which is 6.5 in decimal.

2) An 'understood' binary point is found by shifting the assumed binary point from immediately after the sign bit to immediately after the Nth bit after the sign bit, where N is the number following the 'B'. If the number N is negative, then the assumed binary point is shifted N bits to the left from the sign bit. Thus, for our number 65B4E-1, the understood binary point follows immediately the fourth bit after the sign bit.

3) Now the binary point in the number as converted and the understood binary point in the 7090 word are alined. Then the 36 bits which correspond to the machine word are used as the word generated by this decimal data item. Thus, in our example of 65B4E-1, alining the two points and taking the 36 machine bits produces the octal number 150000000000. Note that two leading zeroes have been inserted before the 1101 produced by the principal and exponent parts - the first zero is the sign bit and the second zero comes from the alining of the decimal points.

In the process of shifting the converted word to position the binary point, significant bits may be shifted past the right-hand end of the word and lost. In this case no error will be indicated. However, if non-zero bits must be shifted past the left-hand end of the word, an error will be indicated (G flag). Thus, the integral part of a fixed point number must be small enough to fit in the number of integral places allowed. Also, if the binary place part is negative, the number must be an appropriately small fraction. For example, the

Following fixed point numbers all specify the same configuration of bits, but not all of them specify the same location for the understood binary point.

```
22.5B5
11.25B4
1125B4E-2
1125.E-2B4
9B7E1
```

All of these fixed point numbers will be converted to the binary configuration whose octal representation is 264000000000.

DECMOD - DECIMAL MODE

Sets the global conversion mode for integers to decimal. This mode may be modified locally by use of qualifiers, but it is reset to decimal, following each card, until a 'SAK' or a 'OCTMOD' occurs.

DETAIL - LIST IN FULL

This pseudo-op reverses the effect of the 'BRIEF' pseudo-op. This is the normal assembly print mode.

EJECT - LISTING PAGE EJECTION

This pseudo-op causes a new page to start in the listing. The 'EJECT' is not printed in the listing.

END - END OF PROGRAM OR OF MACRO DEFINITION

The 'END' card terminates the processing of a deck or of a MACRO definition. For program termination, a location field symbol is defined as the total length of the program, including literals, program common, transfer vector, and any remote assemblies. If the location field is numeric, the program length is reset to this value. If the assembly is absolute and the variable is absolute and the variable field is non-blank, a transition card is punched with the value of the variable field as the transfer address. (In relocatable assemblies, transition cards must be punched with the 'TCD' pseudo-op.) A UMAP program must end physically with an 'END' card.

ENDPGM - END PROGRAM CARD

Used with the 'PGM' pseudo-op to write program cards. 'ENDPGM' turns off the effect of 'PGM', makes the assembly relocatable, and sets the location counter to zero. 'ENDPGM' causes deletion of the automatic program card.

ENTRY - SUBROUTINE ENTRY

This pseudo-op indicates that the program is a subroutine and not a main program. The difference is reflected in the program card. The variable field contains a list of symbols, separated by commas, which are to be considered as entry names to this subroutine. These symbols must all be defined within the program itself. Secondary entries may be obtained by preceding the symbol with a minus sign on the 'ENTRY' card. a main program with additional entries may be obtained by an explicit zero appearing as a symbol on an 'ENTRY' card. This is not the same as an 'ENTRY' card with a blank variable field. 'ENTRY' statements may occur anywhere in the program, and they need not precede the definitions of the symbols which they are naming as entries. 'ENTRY' is undefined if the automatic program card is deleted.

EQU - EQUIVALENCE

The symbol in the location field is assigned the mode and the value of the expression in the variable field. (See the sub-section on PASS 2 symbol definition.)

EQUMAX - EQUIVALENCE TO MAXIMUM SUBFIELD

The location field symbol is given the value and the mode of the subfield of the variable field having the maximum value on PASS 1 processing. (PASS 2 symbol definition does not apply to 'EQUMAX'.) Undefined subfields are treated as absolute subfields with a zero value. Negative subfields are treated as large positive subfields (i. e., -1 is greater than +1). Subfields may be symbolic expressions and are separated by commas. 'EQUMAX' may be followed by one or more 'ETC' cards, but subfields may not be continued from one card to another. In case of equal maximum subfields, the location field symbol is given the mode of the first such subfield.

EQUMIN - EQUIVALENCE TO MINIMUM SUBFIELD

'EQUMIN' is the same as 'EQUMAX' except that the minimum valued subfield of the variable field is used instead of the maximum valued subfield.

ERAS - ERASABLE STORAGE ASSIGNMENT

A block of N+1 locations is reserved in erasable storage, where N is the value of the absolute, pre-defined expression in the variable field. This storage begins at the present value of the erasable storage counter and moves down in core. The erasable storage counter starts at $-1_{10} = 77777_8$. A symbol in the location field is defined as the first (i. e., the highest in core) of these locations. To obtain only one erasable location, the variable field should be left blank or should have a zero value. All erasable assignment must be made with 'ERAS' and 'ERLIST' if UMAP is generating the program card.

ERLIST - ERASABLE LIST

This pseudo-op is the same as the 'ASSIGN' pseudo-op, except that all storage assignment is in erasable storage. Thus,

```
        ERLIST  A,,B
```

Is equivalent to

```
A       ERAS    1
B       ERAS    0
```

ETC - ET CETERA

This pseudo-op allows the extension of the variable field (of certain pseudo-ops) over several contiguous cards. Those pseudo-ops which may be so extended mention this fact in their descriptions. Location field of an 'ETC' is always ignored.

EXECT - EXECUTION OUTPUT CONTROL

Causes UMAP to put out binary card images on the system execution tape. 'EXECT' is ignored if a fatal error has occurred in the program. (See 'NEXECT'.) This pseudo-op acts internal to UMAP only and has no effect on the actual execution of the program.

EXTERN - EXTERNAL NAMES

The variable field of this pseudo-op consists of a set of symbols separated by commas. Each of these symbols is entered into the transfer vector for the program. 'EXTERN' is undefined if the automatic program card is deleted.

FLAGOP - OPTIONAL OPERATION FLAGGING

The variable field has the form 'A,B', where 'A' is the BCD name of an operation in the operation table and 'B' is a single, non-break character. After the occurrence of 'FLAGOP' the operation 'A' is always flagged with the non-fatal flag 'B'. The operation 'A' must be defined when the 'FLAGOP' pseudo-op occurs in PASS 1. At most ten different ops may be optionally flagged during an assembly.

FLAGSY - OPTIONAL SYMBOL FLAGGING

The variable field has the form 'A,B', where 'A' is a symbol and 'B' is a single, non-break character. The symbol 'A' will be non-fatally flagged with the flag 'B' throughout the assembly listing. The symbol 'A' must be defined when the 'FLAGSY' pseudo-op occurs in the program on PASS 1. At most ten different symbols may be optionally flagged in a given assembly. If more than one optionally flagged symbol occurs on a card, then only the flag corresponding to the last symbol to occur on the card is printed.

FUL - FULL MODE PUNCHING

This pseudo-op causes the punching of full 24-word binary cards (i. e., non-relocatable cards). The 7-9 punch is automatically placed in column 1 of such cards, but otherwise the card is exactly the 24 words specified by the program. The automatic program card is deleted by the occurrence of 'FUL'.

HEAD - HEAD SYMBOLS

The 'HEAD' card supplies to the assembly program a single character (punched in column 1 of the 'HEAD' card). Any alphabetic or numeric digit is permissible. Each symbol in the program following the 'HEAD' pseudo-operation is prefixed by this character except when a special indication to cancel the prefix operation is given. A new 'HEAD' pseudo-operation card will replace the prefix character. Thus, several programs having non-unique symbols may be combined by heading each program with a different character.

It is sometimes necessary to make cross-references between the individual programs. To accomplish this, each reference must be written in the following way. Let H be a heading character, and let K be a symbol, in the block headed by H, to which reference is to be made. To refer to K (i. e., to use the value represented by K in an address, tag, or decrement) in a part of the program not headed by H but by J, write,

H$K

The special character $ indicates to the assembly program that K is to be prefixed by H instead of by the prefix J given on the most recent 'HEAD' card.

It is important to note that if use is to be made of the heading feature, all symbols used throughout the program will usually be restricted to five or fewer characters. If any six-character symbols are used, these symbols will not be headed. Some additional remarks are that,

1) A$B is not the same as AB. It is the same as A0000B.
2) A$BCDEF is the same as ABCDEF.
3) 000A, where 0 is zero, is the same as 0A and the same as A.
4) A symbol in an unheaded portion of a program can be referred to from a headed portion by preceding the symbol with a $. The fact that the $ is not preceded by a heading character indicates that reference is to an unheaded section.

An additional feature of the 'HEAD' is the following. If a set of single characters, separated by commas, occurs in the variable field, then all these characters, plus the one in column 1, are saved for headings. Any symbol which is found will be entered in the symbol table once for each separate heading. References within this section are under the primary heading (from column 1) only, but the symbols are defined for all the secondary headings given in the variable field. This eliminates the need to define common symbols in each headed section. The same could be done by making all common symbols 6 characters long. When the next 'HEAD' card is found, all heading characters from the last 'HEAD' card are suppressed.

INDEX -- INDEX SYMBOL DEFINITIONS

Variable field is a list of symbols, separated by commas, which have been defined in the program. Each symbol, along with its definition, is printed in the listing. (This printing is under 'DETAIL' and 'BRIEF' control)

LIST - TURN ON LISTING

This pseudo-op turns on the listing of the assembly of the program, reversing the effects of 'NOLIST'. This is the normal listing mode.

LOC - RELOCATE PROGRAM SEGMENT

This pseudo-op controls the location counter L, over the range of one binary card, without affecting the loading address of that card. 'LOC' is normally used in conjunction with 'ORG' to assemble, with one loading address, program segments which are to be moved in storage before they are executed. The present storage location counter value L is used as the loading address of the binary card, while the contents of the card are assembled as if the location counter had the value N = value of the pre-defined variable field of the 'LOC' card. This effect lasts until the end of the binary card or until the next 'ORG' occurs, which ever comes first.

LOOK - .LOOK CALL

Assembles the same as 'CALLIO .LOOK'. The variable field, except for the first element, is the same as for 'CALLIO'.

MIDDLE - REDUCE ASSEMBLY TIME

The occurence of 'MIDDLE' causes UMAP to change intermediate tapes, rewinding the present intermediate tape. This saves delay time between passes waiting for this tape to rewind. 'MIDDLE' should occur at roughly the middle of the deck. 'MIDDLE' is ignored if it occurs in the first intermediate tape buffer load (i.e., in roughly the first sixty cards of the deck) or if it occurs after the intermediate tape has already been changed.

NEWID - CHANGE OBJECT DECK ID

Beginning with the first non-blank character of the variable field, 8 Hollerith characters are taken from the variable field and used as the ID on the next binary object deck card. The present card is punched when 'NEWID' occurs. The numeric part of this ID, on the right, is incremented by one for each card punched. The 'NEWID' card is not printed on the listing.

NEXECT - EXECUTION OUTPUT CONTROL

Causes UMAP to cease writing binary card images on the system exection tape. The 'NEXECT' acts internal to UMAP only, and does not affect actual execution of the program.

NOBJCT - OBJECT OUTPUT CONTROL

Causes UMAP to cease writing binary card images on the system output tape.

NOCOM - NO COMMENTS

Odd occurrences of this pseudo-op cause the deletion from the assembly listing of all 'REM' cards and all '*' type remark cards. Even occurences cause normal printing control.

NOLIST or UNLIST - NO LISTING

Turns off the printing of the assembly. Only lines which have error flags are printed. All references to symbols or literals are not recorded for the reference table.

NONOP - UNDEFINED OPERATION

Whenever an undefined operation is found, the definition of 'NONOP' is used in place of the operation punched. Normally, this reacts as a 'BSS 1' and causes an 0 flag on the listing. Through use of the 'OPSYN' pseudo-op, the definition of 'NONOP' may be changed to any operation desired, so that the programmer may specify the definition to be used for undefined operations.

NULL - PSEUDO-OP NO-OPERATION

This pseudo-op is printed, but otherwise ignored by UMAP. It may be used in conjunction with 'OPSYN' to nullify the effects of certain operation codes in a given assembly. A symbol in the location field is defined as the present location.

OBJCT - OBJECT OUTPUT CONTROL

Causes UMAP to write card images on the system output tape.

OCT - OCTAL DATA GENERATION

The octal integers in the variable field, separated by commas, are assigned to consecutive locations beginning at L. The first blank to the

right of the variable field indicates that everything which follows is a comment. A symbol in the location field is defined as the location of the octal numbers.

The numbers on an 'OCT' card may have the form 'N' or the form 'NKE', where N is an octal number of 12 or fewer digits, with or without a sign, and E is an absolute mode expression. In the latter case, the number N is shifted M octal places to the left, if M is positive, or M octal places to the right, if M is negative, where M is the value of the absolute mode expression E. Thus, 7K3 is assembled as 000000007000.

In the case of 12 digit octal numbers, the following equivalences hold with respect to the high order bit --

-0 = 4 -1 = 5 -2 = 6 -3 = 7

either form may be used in an assembly. For numbers of less than 12 digits, leading zeroes are supplied. If any error is encountered on an 'OCT' card, a G flag is given.

OCTMOD - OCTAL MODE

Sets the global conversion mode for integers to octal. This mode may be modified locally by use of qualifiers, but it is reset to octal, following each card, until a 'SAK' or a 'DECMOD' occurs.

OPSYN - OPERATION SYNONYM

Defines or redefines the location field symbol as a synonym for the BCD operation code in the variable field. The name in the variable field may be any operation code present in the operation table when the 'OPSYN' occurs on Pass 1. The following operation codes may not function correctly if called by a different name - 'END', 'ETC', 'IRP', 'REM', 'RMT', 'BCT', and 'TITLE'.

ORG - ORIGIN SPECIFICATION

This pseudo-op resets the storage allocation counter to L = value of the pre-defined variable field. This counter is initially zero.

If the assembly is relocatable and the variable field is absolute, then a phase error (P flag) may occur when the next location field symbol occurs. This is due to the incrementing of relocatable symbol values between passes, while absolute origins are unchanged.

If this pseudo-op is written 'ORG*', then the re-origin occurs as described, but the next segment of the program is not included in the computation of the program length. This allows the origining of tables, for example, beyond

the end of the program. This cannot be done with a regular 'ORG', since this would cause incorrect placement of the literals. This mode is reset by the next 'ORG' card.

A location field symbol is defined as the value of the variable field with the mode of the assembly.

PAUSE - .PAUSE CALL

This pseudo-op assembles the same as 'CALL .PAUSE'. The variable field except for the first element is the same as for 'CALL'.

PCC - CONTROL CARD PRINTING

Occurance of 'PCC' causes the printing mode to be switched (off to on or vice versa) for those pseudo-ops which control the printing or assembling of other operation codes, but which do not themselves generate machine words. In 'ON' mode, these pseudo-ops are printed. In 'OFF' mode, these pseudo-ops are deleted from the assembly listing. 'ON' mode is the normal UMAP listing mode for control cards.

PCCOFF - NO CONTROL CARD PRINTING

Deletes printing of control cards - see 'PCC'.

PCCON - PRINT CONTROL CARDS

Causes printing of control cards - see 'PCC'. This is the normal print mode.

PCLIST - PROGRAM COMMON LIST STORAGE

This pseudo-op is the same as the 'ASSIGN', except that all storage assignment is in PROGRAM COMMON. Thus,

```
PCLIST  A,,B
```

is equivalent to

```
A    PGMCOM   0
B    PGMCOM   1
```

'PCLIST' is undefined if the automatic program card is deleted.

PCMORG - PROGRAM COMMON ORIGIN

This pseudo-op allows the origining of parts of a program in PROGRAM

COMMON. Either constants or program segments can be so origined. The origin is taken as the value of the pre-defined, absolute expression in the variable field. References may be made from such sections to other parts of the program. Special binary cards are produced to provide for the loading of these sections. All symbols defined in such sections are of absolute mode. This assembly mode is suppressed upon encountering the next 'ORG' card. If the automatic program card is deleted, 'PCMORG' is treated as a 'ORG*'. In using 'PCMORG', it should be noted that the length of a program segment following a 'PCMORG' is not automatically added to the length of PROGRAM COMMON as computed by UMAP for the program card. Thus, the space needed in PROGRAM COMMON for such a segment must be reserved by the 'PGMCOM' pseudo-op. This can be done symbolically as follows.

```
            PCMORG   START
              .
              .
            (PROGRAM SEGMENT)
              .
              .
LAST        (LAST STATEMENT IN THIS PROGRAM SEGMENT)
            PGMCOM  LAST-START
```

where 'START' may itself be defined by a 'PGMCOM' card.

PGM - PROGRAM CARD

Causes binary cards to have a 12-punch in column 1 -- i.e., program card format. Automatically sets storage location counter to zero. This pseudo-op is used to initiate the generation of a program card by the program rather than by UMAP. Deletes the automatic program card. 'PGM' is equivalent to the sequence

```
9LP     4
ORG     0
```

PGMCOM - PROGRAM COMMON STORAGE ASSIGNMENT

A block of N+1 locations is reserved in PROGRAM COMMON storage, where N is the value of the absolute, pre-defined expression in the variable field. This storage begins at the present value of the PROGRAM COMMON storage counter and moves upward in core. The PROGRAM COMMON storage counter begins at the value of the system relocation constant. A symbol in the location field is defined as the last (i.e., the highest in core) of these N+1 locations and is given absolute mode. To obtain a single location, the variable field should be blank or have the value zero. All PROGRAM COMMON storage must be assigned with 'PGMCOM' and 'PCLIST' if UMAP is generating the program card. 'PGMCOM' is undefined if the automatic program card is deleted.

PRINT - .PRINT CALL

This pseudo-op assembles the same as 'CALLIO .PRINT'. The variable field, except for the first element, is the same as for 'CALLIO'.

PST - PUNCH SYMBOL TABLE

Causes the symbol table to be punched between passes in a special column binary form. These cards may be recognized by a 2 punched in the tag of the first word. The cards may be read on subsequent assemblies by either the 'RST' or the 'BINARY' pseudo-ops. The symbol table deck is sequenced when punched. If an 8-character ID is given in the variable field of the 'PST' card, this ID is used on the symbol table deck. If such ID is not given, then the ID 'SMTBL' is used, with sequence numbering starting at 001.

PUNCH - .PUNCH CALL

This pseudo-op assembles the same as 'CALLIO .PUNCH'. The variable field, except for the first element, is the same as for 'CALLIO'.

PUNLIT - PUNCH LITERALS

Causes suspension of the assembly process in PASS 2 to print the literals on the assembly listing and to punch the literals on binary cards. If 'PUNLIT' does not occur, the literals are printed and punched after the recognition of the 'END' card. Note that 'PUNLIT' has no effect upon where the literals are assigned in storage.

READ - .READ CALL

This pseudo-op assembles the same as 'CALLIO .READ'. The variable field, except for the first element, is the same as for 'CALLIO'.

REF - REFERENCE TABLE SWITCH

Occurrence of 'REF' causes the reference table switch to be changed (off to on or vice versa). In 'ON' mode, references to symbols and to literals are collected for printing in the reference tables. This is the normal reference table mode in UMAP.

REFOFF - REFERENCES OFF

Turns off references to program symbols and to program literals.

REFON - REFERENCES ON

Turns on references to program symbols and to program literals. This is normal assembly mode.

REL - RELOCATABLE PROGRAM

Causes a relocatable assembly with suppression of the automatic program card. Used when a non-standard program card is desired (with 'PGM') or for relocatable override assemblies where no program card is desired.

RELIST - RESTART LIST

Allows continuation of the variable field list structure in a calling sequence after the initial list sequence is broken. Thus, a calling sequence may begin with a 'CALL' or 'CALLIO', continue with 'ETC', break-off for machine coding, then continue with 'RELIST', plus more 'ETC' if desired. Note that the list could not be continued with an 'ETC'. The 'RELIST' must occur first. For example--

```
READ     F,N
LXA      N,1
IOP      B,1
TIX      *-1,1,1
RELIST   C,D,0
```

REM - REMARK

The contents of columns 1-7 and 11-80 are printed on the listing. Columns 8-10 are blanked out.

RESERS - RESET ERASABLE

Causes the erasable storage location counter to be reset to $-1_{10}=77777_8$.

RESPGC - RESET PROGRAM COMMON

Causes the program common storage location counter to be reset to the system relocation constant.

RESTOR - .RSTOR CALL

Assembles as a 'CALLIO .RSTOR'. The variable field, except for the first element, is the same as for 'CALLIO'.

RMT - REMOTE ASSEMBLY

Causes the suspension of the normal assembly process to define all instructions up to the next occurrence of 'RMT' as a remote assembly sequence. All remote assembly sequences are assembled when the 'END' card is found, in front of the program literals, unless they are called for earlier by a 'RMT' with an asterisk in the variable field. Remote sequences are printed where they are found in the deck and under 'PMC' control where they are expanded.

RST - READ SYMBOL TABLE

Causes suspension of the PASS 1 assembly process to read in a symbol table from binary cards which immediately follow the 'RST'. The symbol table cards must conform to the format produced by the 'PST'. The checksum on these cards is checked, and the assembly terminates if a bad checksum is found or if a binary card is found which is not a symbol table card.

The symbols read from this symbol table deck are added to the symbol table. Previous symbol definitions are maintained. Symbol table overflow is possible while reading a symbol table deck. The symbols read in are subject to all heading characters in effect at the time the 'RST' is encountered (see 'HEAD'). However, headings in effect at the time the symbol table was punched (other than a zero head) take effect over headings at the time of reading. If it is desired to completely replace the symbol table by that of the deck, the 'RST' should be preceded by a 'ZST'.

In relocatable programs which include a symbol table reading with either 'BINARY' or 'RST', and in which UMAP is generating the program card, the following restrictions must be observed.

1. No PROGRAM COMMON may be reserved.
2. No transfer vector may be generated.
3. No literals may be used.

In absolute programs, no restrictions exist.

SAK - INTEGER CONVERSION MODE SWITCH

The occurrence of 'SAK' causes the global integer conversion mode to switch to octal, if it is decimal, or to switch to decimal, if it is octal. This mode may be modified locally by qualifiers, but is reset after each card. See also 'OCTMOD' and 'DECMOD'. This mode applies to integers in all fields of a card.

SAVE - .SAVE CALL

Assembles as a 'CALLIO .SAVE. The variable field, except for the first element, is the same as for 'CALLIO'.

SET - SET SYMBOL DEFINITION

Defines the symbol in the location field to have the value and the mode of the expression in the variable field. If the symbol is already in the symbol table, this new definition is given to it without multiple definition. If the symbol is not in the symbol table, then it is entered with this definition. This re-definition occurs in PASS 1 and in PASS 2. See the PASS 2 symbol definition section.

SETTO - .SET CALL

Assembles as a 'CALLIO .SET'. The variable field, except for the first element, is the same as for 'CALLIO'.

SPACE - LISTING SPACING

Causes a spacing of N lines on the listing, where N is the value of the absolute expression in the variable field. If N is greater than 20 or if N is greater than the number of lines remaining on the page, then 'SPACE' is treated as an 'EJECT'. 'SPACE' is not printed in the listing.

SST - SYSTEM SYMBOL TABLE DEFINITION

'SST' allows the incorporation into a program of symbols from the MAMOS SYSTEM SYMBOL TABLE, which is the table of definitions of most system symbols. The variable field of 'SST' is a list of symbols to be defined, separated by commas. Each of these symbols and its definition is placed in the assembly symbol table. If the location field is non-blank, the location field symbol is placed in the assembly symbol table with the definition of the first system symbol in the variable field. All definitions are absolute and non-system symbols are ignored. 'SST' should occur early in the program as normal UMAP PASS 1 processing wipes out the system symbol table in UMAP. 'SST' is ignored if the system symbol table has been clobbered. For any particular symbol in the system symbol table, its definition can be obtained only once. Thereafter that symbol is ignored on 'SST' cards. It is possible to fill up the assembly symbol table while processing a 'SST'. In this case the remainder of the 'SST' card, and all further 'SST' cards, are ignored. An attempt to define a symbol in any other way, however, will cause a fatal error in UMAP. In relocatable programs in which UMAP is generating the program card, system symbols which are defined by an 'SST' and by a 'CALL' will be multiply-defined. Only those system symbols which do not go into the transfer vector should be defined by an 'SST'.

START - START OF PROGRAM

If the variable field is blank, then the present value of the storage

location counter is taken as the first executable statement of the program. If there is a symbol in the variable field, then this symbol is taken as the name of the first executable statement of the program, and the symbol must be defined in the program (but not necessarily before the occurrence of 'START'). If no 'START' occurs in the program, then UMAP assumes that the first executable location in the program is immediately after the transfer vector. 'START' is undefined if the automatic program card is deleted or if the program is a subroutine with no zero name entry.

SYMBOL - SYMBOL TABLE LENGTH ASSIGNMENT

Through use of this pseudo-op, a program may at assembly time adjust the UMAP symbol table and MACRO table lengths. A storage area of about 12,000 locations (this varies with the version of UMAP) contains the symbol table at one end and the macro tables at the other. Normally, the symbol table is defined to contain a maximum of 500 symbols, and the remainder of the 12,000 locations are assigned to the MACRO tables. Thus, the macro tables contain about 11,000 locations since the symbol table is double-entry and requires 1,000 locations. The 'SYMBOL' pseudo-op adjusts the normal settings of these tables as follows. Let 'N' be the value of the absolute, pre-defined expression in the variable field of the 'SYMBOL' card. Then the occurrence of this pseudo-op causes the symbol table to be re-defined to contain a maximum of N symbols--i.e., a maximum length of 2*N locations. The remainder of the 12,000 available locations are defined as the MACRO tables. If the variable field of 'SYMBOL' is blank, then the present symbol table length is taken as the maximum length. 'N' must be at least as great as the number of symbols in the symbol table at the time 'SYMBOL' occurs so that no symbols are lost from the table. Further, 'N' must be small enough that the new symbol table does not overlap the present MACRO definition table. No MACRO definitions may be lost due to adjustment of the tables. Use of the 'ZST' and 'ZMT' pseudo-ops allows the zeroing of these tables before using 'SYMBOL'. If any of the above restrictions are violated, or if 'SYMBOL' occurs in a MACRO, then 'SYMBOL' is non-fatally flagged but otherwise ignored. 'SYMBOL' may occur as often as desired.

SYN - SYNONYM

'SYN' is the same as 'EQU'.

TAPERD - .TAPRD CALL

Assembles the same as 'CALLIO .TAPRD'. The variable field, except for the first element, is the same as for 'CALLIO'.

TAPEWR - .TAPWR CALL

Assembles the same as 'CALLIO .TAPWR'. The variable field, except for the first element, is the same as for 'CALLIO'.

TCD - TRANSFER CARD

Causes the punching of any accumulated binary output followed by the punching of a transfer card. If the assembly is absolute, then the value of the variable field is used as the address of the transfer card.

TITLE - LISTING TITLE

Each page of the listing is headed by a title with a page number. When 'TITLE' occurs, the current page is terminated, a new title is made using columns 14-72 of the 'TITLE' card, and the next page is begun with this new title. To title the first page of the listing, a 'TITLE' card must be the physically first card of the UMAP deck. If an integer appears in the location field of the 'TITLE' card, then the page numbering is set to this value for the next page. 'TITLE' is not printed in the listing.

VFD - VARIABLE FIELD DATA GENERATION

This pseudo-op is primarily useful in constructing tables at translation time. The constituents of the 'VFD' are as follows.

1. A symbol or blanks in the location field
2. The operation code 'VFD'
3. One or more subfields (as described below) in the variable field.

Each 'VFD' generates one or more object words. Each subfield of the variable field generates one or more bits of an object word. Thus, the unit of information for this pseudo-op is the single bit.

The constituents of a subfield are as follows.

1. The Type Letter

   This letter indicates the type of the subfield. Three types are permitted.. Boolean (octal), Hollerith, or symbolic. The corresponding type letters are.....

Boolean (octal)	B or O or K
Hollerith	H or C
Symbolic	No alphabetic character at all

2. The Bit Count

   This is an unsigned decimal integer which specifies how many bits of the object word will be generated by this subfield.

3. The Separation-Character

   A slash (/) is used to separate the bit count from the data item.

4. The Data Item to be converted. The form of the data item depends upon the type of subfield.....

   A. In a symbolic subfield, the data item consists of one UMAP symbolic expression.

   B. In a Boolean (octal) subfield, the data item consists of one octal integer or one UMAP Boolean expression.

   C. In a Hollerith subfield, the data item consists of a string of characters, none of which is comma or blank.

Consecutive subfields are separated by commas. Any number of subfields may be given in a 'VFD'-'ETC' sequence, but no subfield may have a bit count which exceeds 63. Note that a subfield may generate as many bits as desired, but no more than 63 of them will be used. A 'VFD' card may be followed by as many 'ETC' cards as desired, each of whose variable fields corresponds to the above stated restrictions. However, a subfield begun on one card must be terminated on that card. UMAP automatically ends a subfield at the end of a card.

If there is a symbol in the location field of a 'VFD', this symbol is defined as the location into which the first object word generated will be loaded. Location field symbols are ignored on 'ETC' cards.

Successive subfields of the variable field of the 'VFD' are converted and packed to the left to form generated object words. If N is the bit count of the first subfield, then the data item in that subfield is converted to an N-bit binary number. This N-bit binary number is placed in the left-most N bit-positions of the first object word generated. The sign position is here regarded as the first bit-position. If N exceeds 36, the left-most 36 bits of the converted data item form the first generated object word, and the remaining bits of the converted data item are placed in the first N-36 bit-positions of the second generated object word. Each succeeding subfield is converted and placed in the left-most bit-positions remaining after the preceding subfield has been processed. The object words thus generated are assigned to successively higher storage locations. If the total number of bit positions used by all the subfields is not a multiple of 36, then the unused bit-positions at the right of the last generated object word are filled out with zeroes.

The data item in a symbolic subfield is converted as a symbolic expression. Let N be the bit count of the subfield. If the data item as converted occupies more than N bits, only the right most N bits of the converted data item are used. If the data item, as converted, occupies fewer than N bits, then sufficient zero bits are placed at the left of the converted data item to form an N-bit binary number (i.e., the converted data item is right-justified with leading zeroes within its bit count). Neither of these conditions is regarded as an error by UMAP. The asterisk may be used as an element in a symbolic subfield. In this context it

carries the usual meaning of present location. That is, the value of the asterisk will be the location assigned to the generated object word which contains the left-most bit of the converted subfield in which the asterisk appears. Failure to keep this fact in mind may lead to errors, since the bits generated by one subfield may occupy as many as three different generated object words. If the data item is a relocatable expression or an erasable expression, then the subfield must be so situated, relative to other subfields, that its right-most bit coincides with the right-most bit of a generated object word, or with the right-most bit of the decrement portion of a generated object word. This requirement stems from the scheme for handling relocation. A violation of this rule will be flagged by UMAP as a relocation error (R FLAG).

The data item in a Boolean subfield may be an unsigned octal integer of any length. If the bit count of the subfield is 36 or less, the data item may be any valid Boolean expression. Note that an unsigned octal integer is one type of valid Boolean expression. In a Boolean subfield, Boolean mode is always assumed. See the Boolean expression section. Let N be the bit count of the subfield. If the data item, as converted, occupies more than N bits, only the right-most N bits of the converted data item are used. If the data item, as converted, occupies fewer than N bits, then sufficient zero bits are placed at the left of the converted data item to form an N-bit binary number (i.e., the converted data item is right-justified with leading zeroes within its bit count). Neither condition is regarded as an error by UMAP. The B-type, O-type, and K-type subfields are treated exactly the same. The user may use whichever notation is most desirable for him.

The data item in an H-type Hollerith subfield may consist of any combination of characters other than comma or blank. Each character is converted to its six-bit binary-code equivalent. Let N be the bit count for the subfield. If the data item, as converted, occupies more than N bits, only the right-most N bits of the converted data item are used. If the data item, as converted, occupies fewer than N bits, then sufficient six-bit groups of the form 110000 (the BCD code for a blank) are placed at the left of the converted data item to form an N-bit binary number. If N is not a multiple of 6, then some sub-portion of the code for a blank (i.e., some right-most part of 110000) will appear at the extreme left of the N-bit result for this subfield. In other words, the data item is converted as if the left-most character were preceeded by an unlimited number of blanks. If the bit count is not a multiple of 6, then the left-most character used, or the left-most blank used, is truncated from the left. None of these conditions are regarded as an error by UMAP. The above applies to a C-type Hollerith field also, except that leading zeros, instead of leading blanks, are supplied on the left of the converted data item if it occupies fewer than N bits. It is only in this one case that the C-type and H-type Hollerith subfields differ.

If the bit count of any subfield exceeds 63, it is taken as 63, and UMAP will signal an error through the use of a G FLAG. If the bit count of a Boolean subfield exceeds 36, then the data item cannot be a Boolean expression. It can only be an unsigned octal integer.

For example, the statement

```
        VFD     C18/A,H18/A,C18/ABC,H18/ABC,C18/ABCDE,H18/ABCDE
```

generates the three octal words

```
000021606021
212223212223
232425232425
```

and the statements

```
        VFD     18/SYM1-SYM2-10,H18/AB,C18/AB,B6/12345
        ETC     B12/SYM2+SYM3,B24/123456,C30/ABCD,C4/A
SYM1    BOOL    77777
SYM2    EQU     3
SYM3    BOOL    41
```

generate the octal sequence

```
077762602122
002122450043
222324040000
```

ZERO - LOAD ZERO LIST

This pseudo-op is exactly the same as 'ASSIGN', except that at load time zeroes are loaded into all locations defined. Thus,

```
        ZERO    A,,B
```

is equivalent to

```
A       PZE
        PZE
B       PZE
```

ZST - ZERO SYMBOL TABLE

This pseudo-op causes the removal, on PASS 1 only, of all the symbols currently in the symbol table.

EXTENDED MACHINE CODES

All extended machine codes may have a symbol in the location field for reference purposes.

BFT - BOOLEAN OFF-TEST

Assembles as 'LFT' or 'RFT' according as the variable field is a left-Boolean or a right-Boolean expression. Boolean mode is assumed for the variable field evaluation.

BLK - BLOCK PARAMETER

Indicates a block parameter in a non-I/O calling sequence. Requires an address and a decrement.

BNT - BOOLEAN ON-TEST

Assembles as 'LNT' or 'RNT' according as the variable field is a left-Boolean or a right-Boolean expression. Boolean mode is assumed for the variable field evaluation.

BRANCH or BRA - TRANSFER INSTRUCTION

Acts as a transfer when executed. Requires an address. Decrement may be used for storage.

ENDIO - END I/O CALLING SEQUENCE

Terminates an I/O calling sequence. Variable field should be all blank.

FMT - FORMAT SPECIFICATION

Specifies a format for an I/O calling sequence. Address should be the format name. No decrement required.

IIB - INVERT INDICATORS BOOLEAN

Assembles as 'TIL' or 'TIR' according as the variable field is a left-Boolean or a right-Boolean expression. Boolean mode is assumed for the variable field evaluation.

IOBP, IOBPN, IOBT, IOBTN

Used in calling upon low-core I/O control routines.

IOP - I/O PARAMETER

Used to specify either single or block parameters in I/O calling sequences. Requires an address. If block parameter, decrement must be used.

IOTRA

Used in calling low-core I/O control routines.

PAR - NON-I/O SINGLE PARAMETER

Specifies a single parameter in a non-I/O calling sequence. Requires an address. The decrement should not be given.

RIB - RESET INDICATORS BOOLEAN

Assembles as a 'RIL' or as a 'RIR' according as the variable field is a left-Boolean or a right-Boolean expression. Boolean mode is assumed for the variable field evaluation.

SIB - SET INDICATORS BOOLEAN

Assembles as a 'SIL' or as a 'SIR' according as the variable field is a left-Boolean or a right-Boolean expression. Boolean mode is assumed for the variable field evaluation.

SLF - SENSE LIGHTS OFF

When executed, turns off all sense lights. Variable field should be blank.

SLN - SENSE LIGHT ON

When executed, turns on sense light N, where N is the value of the absolute expression in the variable field. N must be greater than zero and less than five.

SLT - SENSE LIGHT TEST

When executed, tests sense light N and skips the next instruction if it is on (also turning off sense light N), where N is the value of the absolute expression in the variable field. N must be greater than zero and less than five.

SWT - SENSE SWITCH TEST

When executed, tests sense switch N and skips the next instruction if switch N is depressed, where N is the value of the absolute expression in the variable field. N must be greater than zero and less than seven.

TAPENR - TAPE NUMBER

Used to specify a tape number in an I/O calling sequence. Address is required. A decrement should not be given.

ZAC - ZERO THE ACCUMULATOR

Causes the accumulator to be zeroed when executed. Requires no variable field.

ZAD or ZSA - ZERO ADDRESS

When executed, causes the address portion of location X to be zeroed out, where X is the value of the variable field.

ZDC or ZSD - ZERO DECREMENT

When executed, causes the decrement portion of location X to be zeroed out, where X is the value of the variable field.

EVEN - FORCE NEXT LOCATION TO BE EVEN

This pseudo-op is used to ensure an even value of the program counter for the data or instruction that follows. It is used primarily with 7094 double-precision instructions. The variable and location fields of the EVEN pseudo-op are blank.

If the program counter is odd when the EVEN pseudo-op is encountered, a binary word containing the instruction AXT 0,0 is generated. Also, for relocatable assemblies, an indication is given in the program card that relocation of the program should be by an even amount, and an extra AXT 0,0 is added following the transfer vector if the number of entries in the transfer vector is odd. If EVEN occurs in a relocatable program, the fourth word of the first program card is made negative. This indication is used by the loader to ensure the program is relocated with its origin at an even location.

COMMON - PSEUDO-OP FOR STORAGE ALLOCATION

This pseudo-op is used to reserve an area of upper core storage for data storage or working space. This pseudo-op is not normally used in programs operating under MAMOS, since the operation ERAS allocates storage in UMAP which is compatible with MAD program erasable storage. The COMMON pseudo-op is provided mainly for compatibility between UMAP and FAP, and storage is allocated in the same manner as described for the FAP pseudo-op COMMON. The difference between

'COMMON' and 'ERAS' pseudo-ops in UMAP are:

1. The 'COMMON' origin is $(77461)_8$ while the 'ERAS' origin is $(77777)_8$
2. 'COMMON N' allocates N locations while 'ERAS N' allocates N+1 locations.

The 'COMMON' and 'ERAS' pseudo-ops can not both be used in the same UMAP program.

COUNT - PSEUDO-OP

This pseudo-op is at present recognized and ignored, and is in UMAP for FAP compatibility. The 'MIDDLE' pseudo-op should be used in UMAP to speed up long assemblies.

LBL - BINARY CARD LABEL

This pseudo-op is provided for FAP compatibility, even though it does not function exactly like the 'LBL' pseudo-op of FAP. The 'LBL' pseudo-op of UMAP functions exactly like the 'NEWID' pseudo-op. That is, the first 8 characters of the variable field is taken as the new label for binary cards.

TTL - LISTING TITLE

This pseudo-op is provided for FAP compatibility, although it does not function in the same manner as the FAP pseudo-op. Except for the inability to title the first page of the listing with 'TTL', this pseudo-op in UMAP functions in the same manner as the UMAP pseudo-op 'TITLE'. The 'TITLE' pseudo-ops of FAP and UMAP are entirely different, but a UMAP assembly of a program containing the 'TITLE' pseudo-op of FAP would result in a correct assembly.

DESIST - STOP ASSEMBLY PROCESS

This pseudo-op causes the assembly process to be suspended. All cards up to the next occurrence of a 'RESUME' pseudo-op are treated as comment cards and printed under 'NOLIST' control.

RESUME - RESUME ASSEMBLY PROCESS

This pseudo-op turns on the assembly process which may have been turned off by a previous 'DESIST' pseudo-op. If already on 'RESUME' is ignored.

OPD - OPERATION DEFINITION

The OPD (Operation Definition) pseudo-operation is used to define a machine operation code. The octal number in the variable field of the OPD pseudo-operation is assigned as the machine operation code definition of the symbol in the location field of the OPD pseudo-operation.

The octal number in the variable field of the OPD pseudo-operation is considered as 36 binary bits which are described as follows.

BITS	MEANING
1 - 12	Operation Code.
13 - 14	Both bits non-zero if indirect addressing is legal.
15	Non-zero if an address is required.
16	Non-zero if a tag is required.
17	Non-zero if a decrement is required.
18	Non-zero if the address does not contain part of the operation.
19	Non-zero for channel commands which may be indirectly addressed.
20	Non-zero for non-transmit channel commands.
21	Non-zero to indicate a machine instruction.
22	Used internally by UMAP.
23	Not used at present.
24 - 36	Remainder of op-code if bit 18 is zero.

If bit 18 is non-zero then bit 35 is non-zero for variable length instructions, and bit 36 is non-zero for sense indicator instructions.

PREFIX CODES

A set of operation codes is provided which allows for the symbolic designation of each of the four parts of a machine word, i.e., the prefix, the decrement, the tag, and/or the address. The operation code determines which prefix is used as follows.

OPERATION CODE	OCTAL PREFIX
PZE or ... or ***	0
PON or ONE	1
PTW or TWO	2
PTH or THREE	3
MZE or FOR or FOUR	4
MON or FVE or FIVE	5
MTW or SIX	6
MTH or SVN or SEVEN	7

The address, tag, and decrement subfields may then be provided in their usual order in the variable field of the instruction. No variable field is required for any of these. Any combination of subfields may be specified. Indirect addressing is <u>not</u> allowed on any of these. All prefix codes may have a symbol in the location field for reference purposes.

MACROS

I. INTRODUCTION

MACRO instructions are sequences of coding which have been given a name and which may have variable parts. Such sequences, once defined in a given program, may be incorporated into that program simply by giving the name of the sequence along with the information to be substituted for the variable parts of the sequence. Pseudo-ops are provided which allow a limited amount of conditional assembly and repetition of segments of the original sequence.

The term MACRO expansion will be used throughout this description. MACRO expansion will mean the following two step process.

1) Determination of the set of arguments given in the MACRO call with the creation of symbols, when necessary, for missing arguments.

2) Generation of UMAP statements, as given in the MACRO definition, with all dummy arguments replaced by the corresponding calling arguments.

The entire process is performed in PASS 1 of UMAP. Step 1 occurs as soon as the MACRO call is recognized as such. Step 2 actually involves a communication process between the MACRO compiler (a sub-section of UMAP) and UMAP proper. The MACRO compiler generates the UMAP statements one at a time. After each statement is generated, control is returned to UMAP, and the generated statement goes through the normal PASS 1 UMAP processing. The assembly process is then interrupted, and control is returned to the MACRO compiler for the next generation step. The replacement of arguments is a purely symbolic process with no checking. Errors are caught during the normal assembly process.

In reading the remainder of this description, it will be important to remember that MACROS are defined and expanded during PASS 1 of UMAP and thus have no more connection with execution time than do other parts of the assembly. The result of a MACRO expansion is a set of UMAP instructions ready to be processed by the normal UMAP assembler, and all instructions throughout this set must meet all the specifications normally imposed by UMAP. As will be seen later, it is the responsibility of the user to insure that the results of MACRO expansions are legitimate UMAP sequences. Absolutely no checking is provided by the MACRO compiler during the expansion process.

II. MACRO definition

The pseudo-op 'MACRO' is used to define a sequence of UMAP instructions as a MACRO. The form of a MACRO definition is

```
NAME    MACRO    A1,A2,...,AN
          .
          R
          .
NAME    END                        or    NAME    ENDM
```

where

NAME	is the name of the MACRO.
a1,A2,...	are the variable parts of the definition, hereafter called the arguments of the MACRO.
R	is a set of UMAP instructions which are to be assembled, with replacement of the variable parts, when the MACRO is called. This set of instructions will hereafter be called the range of the MACRO definition.

The MACRO definition is terminated by an 'END' or 'ENDM' card which has 'NAME' in either the location field or the variable field, or which has a blank variable field and a blank location field. In the latter case, all MACRO definitions in progress (see 3 below) are terminated simultaneously. In the former case, only the MACRO definition whose name occurs in the location field or the variable field is terminated.

The following qualifications and restrictions apply to MACRO definitions.

1) The MACRO name and each argument must be a legal UMAP symbol. Expressions are not allowed in the definition. These symbols need not be distinct from ordinary symbols occurring elsewhere in the program, but they are unrelated. The first blank after column 16 terminates the 'MACRO' card. Parentheses may not be used in a symbol.

2) The name of a MACRO may be the same as the name of some pseudo-op or machine instruction already in UMAP. However, after definition of the MACRO, the MACRO definition will be used for all occurrences of the name in the operation field, and the original definition of the name is lost.

3) MACRO definitions may be nested, that is, the range of one MACRO definition may contain the definition of another MACRO. In such cases, however, MACRO definitions cannot overlap. That is, if a MACRO definition begins inside another MACRO definition, then it must end inside that MACRO definition. Several nested MACRO definitions may end together on a single 'END' card, however. A further restriction also exists. If MACRO definitions are nested, then an inner MACRO is not defined until every MACRO within which it is nested has been called. Thus, if MACRO 'A' is defined inside the definition of MACRO 'B', then MACRO 'A' is undefined until MACRO 'B' has been called at least once. After MACRO 'B' has been called, MACRO 'A' may be called as often as desired. Note that every time MACRO 'B' is expanded, the definition of MACRO 'A' is changed. MACRO 'A' may be called from inside MACRO 'B' (but only after the definition of MACRO 'A' has been expanded at least once, either in the same call or in an earlier call) or from outside MACRO 'B'.

Note, however, that any arguments of MACRO 'B' which occur in 'A', but which do not occur explicitly in the argument list of MACRO 'A', will have been replaced by the corresponding calling arguments from the last call on MACRO 'B'.

4) The definition of a MACRO may include calls for other MACROS, even though they may not be defined at the time of the definition. However, all such MACROS must be defined before the first call for the original MACRO. The programmer must guard against circular definitions. Such definitions cause a loop in the compiler. It is possible for a MACRO to call upon itself, or for MACROS to call upon each other, by using the 'IFF' or 'WHEN' pseudo-ops to control such calls. An example of this will be given later.

5) Those BCD characters which are not legal as part of a MACRO name or argument will be called break characters. The complete list of break characters is as follows.

1) EQUAL	(=)	7) APOSTROPHE	(')
2) PLUS	(+)	8) COMMA	(,)
3) MINUS	(-)	9) BLANK	
4) ASTERISK	(*)	10) LEFT PARENTHESIS	
5) SLASH	(/)	11) RIGHT PARENTHESIS	
6) DOLLAR	($)	12) ALL SPECIAL CHARACTERS	

Any card in the definition is terminated by the first blank in the variable field. In any case, the variable field is terminated with column 72. All break characters in the range of a 'MACRO' (other than blanks or terminal $) behave exactly like commas. Consecutive break characters in an argument string will not cause the insertion of dummy arguments but will simply be skipped.

6) A 'MACRO' card may be extended by use of 'ETC' cards. In such cases, an argument is normally terminated when a card is terminated to split arguments, a dollar sign must be given to indicate that the present card is terminated but the present argument is continued on the next card which must be an 'ETC' card. If a 'MACRO' or 'ETC' card terminates with a blank, it may be followed by an 'ETC', but an argument can't be split. In any case, the maximum number of arguments that a given MACRO may have is 63.

7) Names of MACROS may not be headed, and any heading characters in effect at the time of a MACRO definition are not incorporated as part of the definition.

8) 'REM' cards may be included within MACRO definitions and are scanned for arguments. On 'REM' cards, blanks do not terminate the card, but trailing blanks are not included in the definition. *-Type remark cards and remarks in variable fields in a MACRO definition are not included in the definition.

III. Example

The following definition provides a three- address addition instruction.

```
ADD3    MACRO   A,B,C
        CLA     A
        ADD     B
        STO     C
ADD3    ENDM
```

IV. MACRO call

Once a MACRO has been defined, it is called for insertion into the assembly by the appearance of the name of the MACRO in the operation field with the information to be substituted for the variable parts of the definition in the variable field. The form of the call is

```
S       NAME          E1,E2,....,EN
```

where

S	optional symbol in the location field. If given, it is assigned as the next location to be assigned at the time of the MACRO call.
NAME	name of the MACRO being called.
E1,E2,...	the information to be substituted for the variable parts of the MACRO definition.

The arguments supplied in a MACRO call may be any UMAP expressions. Their correspondence with the dummy arguments of the MACRO definition is determined by their position in the sequence of arguments. Hence, the ith argument in the MACRO call is substituted throughout the MACRO definition for the ith dummy argument in the MACRO definition.

As an example, consider the following call.

```
ADD3   AB-2,C+/K/12,XYZ
```

with the definition given for ADD3 in the last section, the result of this call would be the sequence

```
CLA    AB-2
ADD    C+/K/12
STO    XYZ
```

the substitution process is entirely symbolic. The MACRO compiler simply takes the string of characters which constitute the argument in the call and substitutes this string for the corresponding symbol (dummy argument) in the MACRO definition. The result of this substitution is then assembled by the normal UMAP assembler and must satisfy all UMAP conditions.

In a MACRO call, a pair of parentheses surrounding an expression indicates that everything between the parentheses is to be taken as a single argument in the expansion. For example, the call

```
ADD3   X1,=3,(Z1,1)
```

results in the sequence

```
        CLA     X1
        ADD     =3
        STO     Z1,1
```

Parentheses may be nested in a call. However, whenever parenthesized arguments are found, the outermost pair is stripped off, and everything between them is used as a single argument. Any remaining parentheses must be legally present, since they will occur in the expansion.

Any part of an instruction may be an argument in a MACRO definition. For example, consider the following definition.

```
JUMP    MACRO   TXX,NAME,X
        TXX     X
        NAME
X       SYN     *
JUMP    ENDM
```

In the following expansion, an entire MACRO call will be substituted for the argument 'NAME' above.

```
        JUMP    TPL,(ADD3,(X,1),Y,Z),ZZ
```

The result of <u>two</u> MACRO expansions (one for 'JUMP' and one for 'ADD3') gives the following sequence.

```
        TPL     ZZ
        CLA     X,1
        ADD     Y
        STO     Z
ZZ      SYN     *
```

Care must be taken in substituting for op-codes. If, for example, an argument is named 'TIX', then a substitution will be made for every occurrence of 'TIX' in the MACRO definition.

If arguments are missing from the end of the argument list in a MACRO call, symbols will be created to fill the vacancies. The symbols will have the form '..N', where 'N' is a three digit integer.
For example, consider the call

```
        JUMP    TMI,(ADD3,R,S,T)
```

this results in the sequence

```
        TMI     ..001
        CLA     R
        ADD     S
        STO     T
..001   SYN     *
```

An explicitly empty argument terminated by a comma will be treated as empty (i.e., blank). Created symbols will be supplied only for arguments missing at the end of the argument string. For example, the call

```
ADD3    A,,
```

gives the sequence

```
CLA     A
ADD
STO     ..001
```

One valuable use for created symbols is in making local references within the MACRO definition. Any symbol occurring in a location field in a MACRO definition will also occur in a location field in each expansion of this MACRO. Hence, such a symbol will be multiply-defined in the assembly. To prevent this, all such symbols occurring in location fields in MACRO definitions should be added to the end of the dummy argument string. By not giving these arguments in each call, created symbols will be used for them, and no multiple definitions will occur.

Any heading character in effect at the time of the MACRO call will be applied to all symbols in the resultant sequence.

For convenience in writing MACRO calls, and to allow a more functional notation in such calls, redundant commas may be omitted. Specifically, a comma need not appear before a left parenthesis nor after a right parenthesis in MACRO argument lists. Such commas may be given if desired. For example, the call

```
JUMP    TPL,(ADD3,(X,1),Y,Z),ZZ
```

(as given earlier) could also be written

```
JUMP    TPL(ADD3(X,1)Y,Z)ZZ
```

As an example of the possible functional notation that can be used, consider the example.

```
COS     MACRO   OP,X
        OP
        STO     X
        CALL    COS,X
COS     END
AC      MACRO
AC      END
```

Note that 'AC' is an empty MACRO definition. Then to assemble a sequence of coding whose purpose is to take the cosine of the cosine of the number in the accumulator, the call would be

```
COS(COS(AC))
```

which expands to the sequence

```
STO     ..002
CALL    COS,..002
STO     ..001
CALL    COS,..001
```

It is not necessary to restrict expressions to be substituted into the location field within a MACRO to 6 characters nor those to be substituted into the operation field to 6 characters. One use of this feature is to substitute bodily a whole instruction for such an argument. The only restriction in such substitutions is that the programmer insert in the expression to be substituted the proper blanks so that all resultant expanded instructions have their various fields in the proper columns. Note that blanks within parentheses do not terminate an argument.

Data pseudo-ops (i.e., 'OCT', 'DEC', and 'BCI') may be included in a MACRO definition if desired. If included, such data will be generated in the program for each call on the MACRO. If the variable field of a 'BCI' starts with a digit followed by a comma, the digit and following comma will be stored as the 'BCI' word count, and the number of words indicated will be stored in the definition. The variable field will be scanned beginning after the comma, so that dummy arguments are recognized if set apart by break characters. If a non-numeric other than comma or asterisk starts a 'BCI' variable field, then the variable field scan begins with that character. In this case, the first thing in the variable field should be a dummy argument. On a 'BCI' card, blanks will not stop the variable field scan, but terminal blanks will not be stored in the definition unless they are part of the word count. When 'DEC' cards appear in a MACRO definition, the letters 'E' and 'B' may cause improper definition if used as dummy arguments. Likewise, if 'OCT' cards occur in the MACRO definition, the letter 'K' may cause improper definition if used as a MACRO dummy argument. The variable fields of 'DEC' and 'OCT' cards are scanned, and any dummy arguments will be found if set apart by break characters.

On a MACRO call, it is possible that a single card will not contain the entire MACRO call argument list. To provide for this case, 'ETC' cards may be used to continue the argument list over several cards. The following conventions hold for 'ETC' cards in a MACRO call --

1) Each card is terminated by the first blank column which is not within parentheses, or by a dollar sign.

2) An argument is not assumed to be terminated when a card is terminated. Hence, an argument may be split between two cards.

3) Any blanks within parentheses are considered part of the argument. Hence, if such an argument is split between two cards, all remaining blanks on the first card will be retained as part of

the argument. A dollar sign may be used to indicate that the argument is continued on the next card and that the remainder of the present card should be ignored.

4) If a MACRO call or a following 'ETC' card ends with a dollar sign, then that card must be followed by another 'ETC' card. If the MACRO call or a following 'ETC' card terminates with a blank, then it may be followed by an 'ETC' card if needed.

V. IRP Pseudo-Operation

The indefinite repeat pseudo-op 'IRP' is used in pairs within a MACRO definition to begin and end a block of instructions which are to be repeated an indefinite number of times at the time the MACRO is expanded. The form of such a block is

```
IRP     A
 .
 B
 .
IRP
```

where

A	=	dummy argument in MACRO definition
B	=	block of instructions to be repeated

The block 'B' will be repeated once for each subfield of the argument 'A' given in the MACRO call, and on each repetition the argument 'A' will be replaced by the current sub-argument given in the call. 'A' must be a UMAP expression defined before appearance of the instruction in a MACRO call. For example, the definition

```
SUMSQ       MACRO   T,B,K
            STZ     T
            IRP     B
            LDQ     B
            FMP     B
            FAD     T
            STO     T
            IRP
            CALL    SQRT,T
            STO     K
SUMSQ       END
```

Defines a sequence which computes the square root of the sum of the squares of the subarguments of 'B'. The call

```
SUMSQ       A,((X,1),(Y,2)),A+1
```

results in the sequence

```
            STZ     A
            LDQ     X,1
            FMP     X,1
            FAD     A
            STO     A
            LDQ     Y,2
            FMP     Y,2
            FAD     A
            STO     A
            CALL    SQRT,A
            STO     A+1
```

The following restrictions and additions apply to the use of the 'IRP' pseudo-op.

1) 'IRP' may occur only inside a MACRO definition. If it occurs elsewhere in a program, it is treated as an undefined operation code.

2) An 'IRP' can't occur explicitly in the range of another 'IRP'. However, a MACRO called from within an 'IRP' range may itself contain other 'IRP' pairs.

3) An 'IRP' on an empty argument or an 'IRP' on a blank variable field causes the skipping of all instructions in the 'IRP' range.

4) An 'IRP Q', where 'Q' is not a dummy argument in the definition, causes the skipping of all instructions in the 'IRP' range.

5) An 'IRP' sequence can't occur inside a remote (see 'RMT' pseudo-op) assembly segment.

Thus, for example, the call

```
SUMSQ     A+5,,A+6
```

will generate the sequence

```
STZ       A+5
CALL      SQRT,A+5
STO       A+6
```

VI. SKIP Pseudo-Operation

The 'SKIP' pseudo-op allows a limited type of skipping within a MACRO definition at the time of expansion of the MACRO. Normally, it is used in conjunction with the 'IFF' or 'WHEN' pseudo-ops (described below). The 'SKIP' pseudo-op will have one of two alternate forms.

1) SKIP With blank variable field.
When this form of 'SKIP' is encountered, the expansion of the MACRO is terminated immediately, i.e., all remaining instructions in the MACRO are deleted.

2) SKIP P P = UMAP symbol or unsigned decimal integer.
If P is a UMAP symbol, then all instructions up to, but not including, the one with P in its location field are skipped. If P is an integer, then the next P statements in the MACRO definition are skipped. P is assumed to be in decimal mode. In either case, the following restrictions hold.

   A) If 'SKIP P' occurs in the range of an 'IRP', then the SKIP terminates as above or with the second 'IRP', whichever comes first.

   B) If 'SKIP P' occurs outside the range of an 'IRP', then the SKIP can't end within an 'IRP' range. When such a situation occurs, the remainder of the 'IRP' range is skipped automatically. A 'SKIP' can, however, skip over any number of complete 'IRP' ranges.

   C) An 'ETC' pseudo-op is always ignored in the skipping process. Thus, if the skipping is done with a count (i.e., 'P' is an integer), then the count should not include any 'ETC' cards which are to be skipped.

   In all cases a skip is in the forward direction. A SKIP cannot return to an earlier part of the MACRO definition. If the end of a MACRO definition is encountered during a SKIP, then the MACRO expansion is terminated.

The 'SKIP' pseudo-op can only occur within MACRO definitions. If it occurs elsewhere in a program, it is treated as an undefined operation code.

VII. IFF Pseudo-Operation

The 'IFF' pseudo-op provides for the conditional assembly of segments of a MACRO definition at the time of the MACRO expansion. The 'IFF' has two forms:

```
FORM 1      IFF   P,A,B
FORM 2      IFF   P,A,B,S
```

where

P	UMAP expression
A	UMAP symbol
B	UMAP symbol
S	UMAP symbol or unsigned decimal integer

To describe these two forms, the following definitions will be needed.

Q	Q=1 if the value of the expression P is non-zero. Q=0, otherwise.
R	R=1 if A and B are identical (i.e., if A and B are the same symbol). R=0, otherwise.

Note that P is evaluated in PASS 1. Undefined expressions are given the value zero. Literals are undefined in PASS 1. P may contain dummy arguments and these will be substituted before the evaluation of P.

With the definitions of Q and R, the two forms of 'IFF' may now be described.

Form 1 IFF P,A,P

The instruction immediately following the 'IFF' statement is included in the MACRO expansion if and only if Q and R have the same value. If Q and R do not have the same value, then the statement following the 'IFF' is deleted from the MACRO expansion.

Form 2 IFF P,A,B,S

This form of the 'IFF' statement is equivalent to the two statements

```
IFF    P,A,B
SKIP   S
```

The restrictions upon skipping with respect to 'IRP' ranges (see VI) apply to this form of the 'IFF', also. Note further that 'ETC' cards are ignored in skipping mode. Thus, whenever a SKIP with a count is initiated by an 'IFF' in any of its permissible forms, the count should not include any 'ETC' cards which are to be skipped.

Note the statement above that 'A' and 'B' are treated as symbols, i.e., the value of each is the octal code for the BCD representation of the symbol. If an expression occurs, the last symbol in the expression, or the final digits of the last number in the expression, are used as the symbol 'A' or 'B'. If literal occurs, the symbol used is the equal sign (=). To insure proper functioning of the 'IFF' pseudo-op, both 'A' and 'B' should be single symbols and not expressions.

As an example, redefine the MACRO JUMP to be

```
JUMP        MACRO       TXX,X,N
            TXX         X
            NAME
            IFF         1,N,1
X           SYN         *
JUMP        END
```

With this definition, the 'SYN' card is included in the expansion only if the final argument is a '1'. Thus, the call

```
            JUMP        JPL,(CLA A,1),Y
```

generates the sequence

```
            TPL         Y
            CLA         A,1
```

while the call

```
            JUMP        M,(STO B),Z,1
```

generates the sequence

```
            TMI         Z
            STO         B
Z           SYN         *
```

The 'IFF' may also be used to allow circular definitions within MACRO in this case, the 'IFF' is used to eventually terminate the apparent circularity. For example, consider the following MACRO which calls upon itself.

```
TXITAB      MACRO       A,B,C
            TXI         A,B,C
            TXI         A,B,-C
            IFF         C/2
            TXITAB      A,B,C/2
TXITAB      END
```

The MACRO call

```
TXITAB    X,Y,8
```

generates the sequence

```
TXI       X,Y,8
TXI       X,Y,-8
TXI       X,Y,8/2
TXI       X,Y,-8/2
TXI       X,Y,8/2/2
TXI       X,Y,-8/2/2
TXI       X,Y,8/2/2/2
TXI       X,Y,-8/2/2/2
```

Note that in the absence of the 'IFF C/2' statement, the MACRO compiler would have gone into an infinite loop in trying to expand this MACRO call. Since C/2 = 0 when C = 1 (due to integer division), the 'IFF' eventually skips the MACRO call 'TXITAB A,B,C/2', thus terminating the MACRO. Had there been additional instructions in the MACRO definition following this call, these would now be expanded, once for each time the MACRO called upon itself, each time using the arguments for the specific call upon the MACRO. Thus, the entire MACRO is expanded once for each call made upon it in this recursive loop.

The reader should be able to verify the following interpretations of 'IFF' statements.

A) `IFF  0,A,B` — Assemble the next instruction if A and B are different symbols.

B) `IFF  1,A,B,S` — Skip to the statement labeled S, if S is a symbol, or skip S statements, if S is an integer, if A and B are the same symbol.

C) `IFF  Z/N` — Assemble the next instruction if Z is greater than N

D) `IF  Z/N,,,S` — Skip to the statement labeled S, if S is a symbol, or skip S statements, if S is an integer and if Z>N.

More generally, in terms of the variable Q defined above, the interpretation of the two forms of the 'IFF' pseudo-op are as follows.

Form 1 — `IFF  P,A,B`

A) Q = 0 — Assemble the next instruction if A and B are different symbols.

B) Q = 1 — Assemble the next instruction if A and B are the same symbol.

Form 2 IFF P,A,B,S

A) Q = 0 Skip to the statement labeled S, if S is a symbol, or skip S statements, if S is an integer and if A and B are different symbols.

B) Q = 1 Skip to the statement labeled S, if S is a symbol, or skip S statements, if S is an integer and if A and B are the same symbol.

The 'IFF' pseudo-op may occur only in MACRO definitions. If it occurs elsewhere in a program, it is treated as an undefined operation code.

VIII. MACRO Qualifiers

A set of seven qualifiers are available for use in Macro CALLS and definitions. Three of these qualifiers (/CRS/, /MAC/, and /MI/) may be used only in the 'IFF' pseudo-op as described below. Anywhere else, they are treated as undefined qualifiers. A fourth qualifier (/NS/) can occur only in the 'IFF' or 'WHEN' pseudo-ops. Anywhere else, it is undefined. The remaining three qualifiers (/I/, /N/, and /P/) may actually occur almost anywhere (i.e., are treated essentially as the other qualifiers of UMAP) but are primarily available for use in MACRO calls and definitions.

The description of the qualifier structure and the restrictions on the manner in which qualifiers may occur in an expression as described previously apply in the use of any of the above seven qualifiers. Failure to comply with these restrictions will cause a qualifier to go unrecognized (which will usually result in an error of some other type in evaluating the pertinent expression) or to be undefined.

A) /I/ Indirect Address Qualifier

If the /I/ qualifier occurs anywhere in a variable field, it indicates that the operation code for this card is to be indirectly addressed. For example, the MACRO call

```
ADD3    (/I/A,1),B,C
```

results in the sequence

```
CLA     /I/A,1
ADD     B
STO     C
```

which is equivalent to the sequence

```
CLA*    A,1
ADD     B
STO     C
```

care must be taken, when using /I/ in Macro calls, that indirect addressing is given only to those op-codes for which it is legal.

B) /CRS/ Created Symbol Qualifier

The /CRS/ qualifier may occur only in the second argument of an 'IFF' statement. If it occurs anywhere else, it is treated as an illegal qualifier.

With the /CRS/ qualifier, the third argument of the 'IFF' statement is not used, and hence must be deleted by the programmer. Using the notation developed in VII, the two forms of the 'IFF' using /CRS/ are as follows.

Form 1 IFF P,/CRS/A

1) Q = 0 — Assemble the next instruction if A is not a created symbol.

2) Q = 1 — Assemble the next instruction if A is a created symbol.

Form 2 IFF P,/CRS/A,S

1) Q = 0 — Skip to the statement labeled S, if S is a symbol, or skip S statements, if S is an integer and if A is not a created symbol.

2) Q = 1 — Skip to the statement labeled S, if S is a symbol, or skip S statements, if S is an integer and if A is a created symbol.

Note that in skipping with a count (S an integer), 'ETC' cards are ignored and hence should not be included in the count.

C) /MAC/ MACRO Name Qualifier

The /MAC/ qualifier may occur only in the second argument of an 'IFF' statement. If it occurs anywhere else, it is treated as an illegal qualifier.

With the /MAC/ qualifier, the third argument of the 'IFF' statement is not needed, and hence it must be deleted by the programmer. Using the notation VII, the two forms of the 'IFF' using /MAC/ are as follows.

Form 1 IFF P,/MAC/A

1) Q = 0 — Assemble the next instruction if A is not a MACRO name.

2) Q = 1 — Assemble the next instruction if A is a MACRO name.

Form 2 IFF P,/MAC/A,S

1) Q = 0 — Skip to the statement labeled S, if S is a symbol or skip S statements, if S is an integer and if A is not a MACRO name.

2) Q = 1 — Skip to the statement labeled S, if S is a symbol, or skip S statements, if S is an integer and if A is a MACRO name.

Note that in skipping with a count (S an integer), 'ETC' cards are ignored and hence should not be included in the count.

D) /MI/ MACRO Indirect Qualifier

The /MI/ qualifier can occur only in the second argument of an 'IFF' statement. If it occurs anywhere else, it is treated as an illegal qualifier.

With the /MI/ qualifier, the third argument of the 'IFF' statement is not needed, and hence must be deleted by the programmer. Using the notation developed in VII, the two forms of the 'IFF' using /MI/ are as follows.

Form 1 IFF P,/MI/

1) Q = 0 — If the call for the MACRO being expanded was not indirectly addressed, assemble the next instruction.

2) Q = 1 — If the call for the MACRO being expanded was indirectly addressed, assemble the next instruction.

Form 2 IFF P,/MI/,S

1) Q = 0 — If the call for the MACRO being expanded was not indirectly addressed, skip to the instruction labeled S, if S is a symbol, or skip S statements, if S is an integer.

Note that in skipping with a count (S an integer), 'ETC' cards are ignored and hence should not be included in the count.

E) /NS/ No Skip Qualifier

The /NS/ qualifier may occur only in the second argument of an 'IFF' statement or the second argument of a 'WHEN' statement. If /NS/ occurs anywhere else in a program, it is treated as an undefined qualifier.

The /NS/ qualifier has the effect of nullifying any skipping in the MACRO definition which might arise from the 'IFF' or 'WHEN' statement in which it occurs. The effect of /NS/ lasts only for the duration of the 'IFF' or 'WHEN' card on which it occurs.

Thus

```
IFF   P,/NS/A,B,S
```

is equivalent to

```
IFF   P,A,B
```

and

```
WHEN  A,/NS/R,B,S
```

is equivalent to

```
WHEN  A,R,B
```

This makes possible the substitution of arguments of the form (V,T) for 'B' without changing the original meaning of the 'IFF' or 'WHEN' statements.

F. /P/ and /N/ Print Control Qualifiers

These qualifiers are supplied to provide the functions normally served by a 'P' or a 'N' in column 7 of a card. Thus, the occurrence of /P/ in a statement causes that statement to be printed regardless of the print control mode in effect (due to such pseudo-ops as 'NOLIST', 'PCC', or 'PMC'). Conversely, the occurrence of a /N/ in a statement causes that statement to be omitted from the listing unless an error is detected while analyzing the card. /P/ and /N/ can occur anywhere that qualifiers are normally permitted. These qualifiers are supplied because column 7 of a card in a MACRO definition is not preserved for the MACRO expansion, and thus the 'P' and 'N' conventions of column 7 can't in general be used. Only in the particular case in which a MACRO argument occurs in a location field and a complete UMAP statement is substituted for that argument can one use the column 7 'N' and 'P' controls. An example of the /N/ qualifier is given in the MACRO 'REP' defined in sub-section IX. The effect of a /P/ or a /N/ is carried over to any following 'ETC' cards where such are legal.

G) Example Usage

The following examples, while not particularly useful MACROS, will demonstrate some of the qualifiers discussed above.

```
QUAL1       MACRO   P,B,A
            IFF     P,/CRS/A
            TXL
            TXI
            IFF     P,/MAC/B
            TXL
            TXI
            IFF     P,/MI/
            TXL
            TXI
QUAL1       END

QUAL2       MACRO   P,B,A
            IFF     P,/CRS/A,S1
            TXL
S1          TXI
            IFF     P,/MAC/B,S2
            TXL
S2          TXI
            IFF     P,/MI/,S3
            TXL
S3          TXI
QUAL2       END
```

The reader should verify that the following calls and generated sequences are paired.

A)	QUAL1	1,X,Y	TXI
			TXI
			TXI
B)	QUAL1*	1,QUAL2	TXL
			TXI
			TXL
			TXI
			TXL
			TXI
C)	QUAL1	C,X,Y	SAME AS B)
D)	QUAL1*	0,QUAL2	SAME AS A)
E)	QUAL2	1,X,Y	SAME AS B)
F)	QUAL2*	1,QUAL1	SAME AS A)
G)	QUAL2	0,X,Y	SAME AS A)
H)	QUAL2*	0,QUAL1	SAME AS B)

IX. WHEN Pseudo-Operation

The 'WHEN' pseudo-operation provides for the conditional assembly of segments of a MACRO definition at the time of the MACRO expansion. The 'WHEN' pseudo-op has one of two alternate forms.

```
Form 1   WHEN  E,R,F
Form 2   WHEN  E,R,F,S
```

where

E UMAP Expression
R a relation -- R must be one of the following

.E.	Equal to
.NE.	Not equal to
.GE.	Greater than or equal to
.G.	Greater than
.LE.	Less than or equal to
.L.	Less than

F UMAP Expression
S UMAP symbol or unsigned decimal integer

To describe this pseudo-op, the following definitions are made.

V = Value of the expression E
W = Value of the expression F

where the expressions E and F are evaluated on PASS 1 of UMAP. (Undefined expressions are given the value zero--literals are undefined on PASS 1.) Now define B as

B = 0 If the Boolean expression V R W is false.
B = 1 If the Boolean expression V R W is true.

The two alternate forms of 'WHEN' then act as follows.

Form 1 WHEN E,R,F

B = 0 Do not include the next statement in the Macro definition in the present expansion of the MACRO..
B = 1 Do include the next statement in the MACRO definition in the present expansion of the MACRO.

Form 2 WHEN E,R,F,S

B = 0 Continue the MACRO expansion with the next statement in the MACRO definition.
B = 1 If S is a symbol, skip all statements in the MACRO definition up to, but not including, the one labelled S. If s is an integer, skip the next S statements in the MACRO definition,

then continue the MACRO expansion. The restrictions upon skipping, with respect to 'IRP' ranges (see VI), apply here also. If the end of the MACRO definition is encountered while in skipping mode, the MACRO expansion is terminated for this call. Note further that 'ETC' cards are ignored in skipping mode so that when skipping by a count (i.e., when S is an integer) any 'ETC' cards to be skipped should not be included in the count.

The 'WHEN' pseudo-op is treated as undefined if it occurs exterior to a MACRO definition. It is also treated as undefined if any one of the first three arguments is not given or if a relation 'R', other than those listed above, is given.

As examples, the MACROS 'JUMP' and 'TXITAB' (see VII) may be redefined using 'WHEN' instead of 'IFF' as follows.

```
JUMP     MACRO    TXX,NAME,X,N
         TXX      X
         NAME
         WHEN     N,.E.,1
X        SYN      *
JUMP     END

TXITAB   MACRO    A,B,C
         TXI      A,B,C
         TXI      A,B,C
         WHEN     C/2,.NE.,0
         TXITAB   A,B,C/2
TXITAB   END
```

The calling sequences for these MACRO definitions are the same as given in VII.

Note that, in contrast with the 'IFF' pseudo-op, only values of symbols or expressions are used in the 'WHEN' pseudo-op, as opposed to the octal BCD representations of such symbols.

Note also that the macro qualifiers /MAC/, /MI/, and /CRS/ (see VIII) may be used only in the 'IFF' pseudo-op. They can't be used with the 'WHEN' pseudo-op. The /NS/ qualifier may, however, be used in the second argument of the 'WHEN' pseudo-operation.

As a final example, consider the following MACRO definition.

```
REP      MACRO    A,B,C
C        SET      /N/A-1
         IRP      B
         B
         IRP
         WHEN     C,.E.,0,1
         REP      C,(B),C
         END
```

This MACRO expands the sequence 'B' of instructions 'A' times. Thus, the call

```
REP     3,((LGL 3),(ALS 3))
```

results in the sequence

```
LGL     3
ALS     3
LGL     3
ALS     3
LGL     3
ALS     3
```

Note that the /N/ qualifier prevents printing of the 'SET' in the expansion. 'REP' is another example of a MACRO definition which calls upon itself. However, note the use here of the 'SET' pseudo-operation to perform the actual counting of the number of expansions made. This, for several internal reasons, is better than the approach used before with the MACRO 'TXITAB' since the symbolic arguments in the repeated call do not become longer (and hence do not require more room, either in the MACRO tables or in the card images generated) and all expressions are easier to analyze. Note, too, that a third argument 'C' is added and is used as a counter. 'C' is a created symbol in the above expansion, since no corresponding call argument is given. This is the usual procedure in constructing counters. Regular symbols may be used if desired. This procedure allows several MACRO definitions to simultaneously count and to communicate among themselves. One final point should be noticed in this MACRO definition. Note that in the statement

```
REP     C,(B),C
```

The parentheses are put back on the set of sub-arguments in 'B'. This is necessary since these parentheses are stripped off when the argument is picked up. Note also that the third argument is given here. This prevents the use of a new created symbol for every call which 'REP' makes upon itself.

X. RMT Pseudo-Operation

MACRO instructions may require the assignment of temporary storage locations, the definition of constants, or other storage allocation. Such storage may be set aside within the MACRO definition, in which case it must be bypassed by transfer instructions, or the programmer may keep track of such requirements for each MACRO instruction and provide the required definitions wherever convenient within the program. The pseudo-op 'RMT' (standing for remote) provides a means by which such storage requirements may be automatically handled by UMAP after the completion of the rest of the assembly. Any instruction cards which occur between two 'RMT' cards (with blank variable fields) will be saved in storage and not assembled until either a 'RMT *' or a normal UMAP 'END' card is encountered. In either case, all cards saved for remote insertion will be inserted into the program and assembled at that point. The various remote sequences will be assembled in the order in which they were originally found. Such remote sequences may include any UMAP operation codes, including calls for MACROS. Such remotely expanded MACROS may themselves include remote sequences. When all remote sequences have been generated, UMAP will go on to the card following the 'RMT *' or will terminate PASS 1 if the remote assembly was called by an 'END' card. In the case of 'RMT *', additional remote sequences may occur after it in the assembly. Such additional sequences will be assembled when the next 'RMT *' or the 'END' card is encountered.

Remote sequences may be defined external to a MACRO definition, and the above discussion still applies. 'RMT *' can't occur within a MACRO definition, nor can an 'IRP' occur within a 'RMT' sequence. 'RMT' sequences should be used sparingly, for they require storage space (taken away from the MACRO tables) and greatly lengthen assembly time. An overflow of the MACRO tables during PASS 1 of UMAP, due either to MACRO definitions and calls or to 'RMT' sequences, causes immediate suspension of the assembly.

In using remote sequences, care must be taken with respect to 'HEAD' and 'SAK' modes. Remote assemblies are subject to the heading character and 'SAK' mode in effect at the time of their assembly and not to those in effect at the time of their definition as a remote sequence. Note that if a 'ZMT' occurs in a program, all remote sequences defined previous to it are lost. Such sequences should be expanded with a 'RMT *' before the 'ZMT'.

XI. Symbol Concatenation

An additional feature available to MACRO definitions is the capability of concatenating arguments to arguments or to other parts of the MACRO definition. The user is warned that this is a capability fraught with danger for many reasons. With the use of concatenation, it is extremely easy to generate in a MACRO expansion expressions which are devoid of all meaning to UMAP or symbols which do not correspond to UMAP restrictions. In any given instance, it is advisable not to use concatenation if other means are available.

The concatenation operator is the apostrophe ('). Its occurrence indicates that its two operands are to be concatenated and henceforth considered as a single symbol. It is the programmers responsibility to ensure that the results of such concatenation are legal UMAP symbols or expressions.

For example, consider the following MACRO definition.

```
OUT   MACRO   A,B,C,D,E,F
      PRINT   F,A,...,A+B-1,0
      RMT
F     BCI     *,1H4,C'H'D,B'E*
      RMT
OUT   END
```

The call

```
      OUT     A1,10,7,(X(I) = ),I10
```

results in the instruction

```
      PRINT   ..001,A1,...,A1+10-1,0
```

with a remote assembly of the format

```
..001 BCI     *,1H4,7HX(I) = ,10I10*
```

The concatenation operator is not effective if it occurs in a location field or if it occurs in an operation field.

XII. NOCRS Pseudo-Operation

The occurrence of the 'NOCRS' pseudo-op suppresses the generation of created symbols for arguments missing from the end of a MACRO call. After the occurrence of 'NOCRS', all missing arguments are treated as explicity empty arguments.

It should be noted, in using 'NOCRS' and 'ORGCRS', that created symbols are generated at the time a MACRO call is found and before any part of the expansion for this call occurs. Thus, whether or not symbols are created for missing arguments in a MACRO call depends upon whether or not a 'NOCRS' is in control at the time the MACRO call occurs. A MACRO definition cannot affect this mode for its own arguments, but it can do so for MACROS which are called from within the definition. Thus, suppose the following situation occurs.

```
M1          MACRO     A1,A2
            NOCRS
              .
              .
              .
            M2        A2
              .
              .
              .
            ORGCRS
            END
M2          MACRO     B1,B2
              .
              .
              .
            PZE       B1,,B2
              .
              .
              .
            END
```

Now suppose the call

```
            LOC       M1        C
```

is given. As the second argument of 'M1' is missing, a created symbol (say '..001') is generated for it. Therefore, the call for 'M2' is expanded (from the 'M1' definition) as

```
                      M2        ..001
```

Note that the occurrence of 'NOCRS' within 'M1' does not affect the created symbols in expanding 'M1'. The above call for 'M2' is missing the second argument, but the occurrence of 'NOCRS' has turned off the generation of created symbols. Thus, this argument is treated as empty. Then, the 'PZE' of the 'M2' definition expands as

```
                      PZE       ..001,,
```

Note that within 'M2' the first argument is replaced by '..001' and appears

to have been missing in the 'M2' call. If the argument 'B2' occurs in 'M2' in an 'IFF' statement and following the /CRS/ qualifier, then it will cause that action to be taken which represents the case in which 'B2' is a created symbol. The MACRO compiler can not distinguish between the created symbol generated in 'M1' (and passed along via the MACRO arguments to 'M2') and the created symbol generated in 'M2' itself.

XIII. ORGCRS Pseudo-Operation

The 'ORGCRS' pseudo-op serves two functions. First, 'ORGCRS' may be used to initiate the creation of symbols in MACRO expansions. In this case, the variable field will usually be blank (but may be of the form 'CNNN' as described below), and the created symbol process begins with the first one not used previously. Second, 'ORGCRS' may be used to re-origin the numeric part of created symbols or to change the form of created symbols. If the symbol 'CNNN' occurs in the variable field, where 'C' is any non-break character and 'NNN' is a three digit integer, then the next created symbol will have the numeric part 'NNN+1' with the leading characters '.C' (instead of '...'). (See the last paragraph of XII for further comments on the effects of 'ORGCRS'.)

XIV. PMC, PMCON, PMCOFF, NOMAC Pseudo-Operations

Normally, MACRO expansions are printed in full except that the 'SKIP', 'WHEN', 'IFF', and 'IRP' pseudo-ops are never included in this printing. When the 'PMC' pseudo-operation occurs, this MACRO printing mode flips from on to off or vice versa ('PMC' is a binary switch). In the 'ON' mode, the complete expansion of each MACRO is printed. This is the normal print mode for MACROS. In the 'OFF' mode, only the MACRO call is printed in the listing. 'PMCON' always turns on the printing of MACRO expansions, while 'PMCOFF' always turns this printing off. Odd occurrences of the 'NOMAC' pseudo-operation cause the deletion from the assembly listing of MACRO definitions and remote sequence definitions. Even occurrences reverse this setting. This pseudo-op has no effect upon the listing of MACRO and remote sequence expansions.

XV. ZMT Pseudo-Operation

This pseudo-operation causes the removal of all MACRO definitions and/or remote sequences from the MACRO tables on PASS 1 processing. All MACROS defined prior to a 'ZMT' will be undefined operation codes subsequent to it (and not subject to the definition of 'NONOP'). 'ZMT' may not occur within a MACRO, but it may occur as often as desired in an assembly. This pseudo-op may be used to prevent MACRO table overflows during PASS 1 processing. Note that the programmer may also accomplish this objective through the use of the 'SYMBOL' pseudo-operation. 'ZMT' does not reset the created symbol count. See 'ORGCRS'.

XVI. MACRO Depth Number

When MACRO expansions are printed, a depth number is printed, for each instruction in the expansion, in columns 111, 112, and 113 of the listing. This is an octal number, and it indicates the nesting depth of the MACRO calls at the time of the expansion. There is no set limit to this nesting. MACRO calls may be nested as deep as desired, except that MACRO table overflows may occur for great nesting depths.

XVII. MACRO Errors

If there is an error on a 'MACRO' card (e.g., a blank location field) or in a MACRO call (e.g., more left parentheses than right parentheses), a 'U' flag results and the definition is not entered or the call is not expanded. If an 'ETC' card incorrectly occurs in a MACRO definition or a MACRO call, an 'E' flag results and the definition is not entered or the call is not expanded. Unfortunately, these errors are rather obvious and are the only ones which prevent the definition entry or the call expansion. The more common MACRO errors result in incorrect sequences of coding, sometimes with no fatal errors so that the assembly appears to be successful. This latter type of error results especially from incorrect arguments in a correct call (e.g., the wrong sequence) or from an incorrect splitting of arguments between one card and a following 'ETC' card in either a MACRO definition or a MACRO call. To prevent the latter, it is advisable to always use the '$' convention in splitting arguments between cards. Quite often, however, errors in a MACRO call or definition result in UMAP sequences which give rise to the usual error types -- illegal qualifier or literal, undefined symbol or op-code, multiply-defined symbol, etc. Other than a careful analysis of the MACRO definition and the MACRO call, possibly with a hand simulation of the call, there is no ready-made procedure for determining why a MACRO did not give the desired coding sequence. The best procedure to follow, especially if the MACRO definitions are relatively complicated, is to test the MACRO definitions with all types of calls, for which they should work, in a separate assembly.

XVIII. Further MACRO Examples

A. A three address addition MACRO

To facilitate the writing of a program, it may be convenient to define the numerical operations ('ADD', 'SUB', 'MPY', 'DVP', 'FAD', 'FSB', 'FMP', 'FDP') as three-address MACROS. The specific example of a three-address floating addition MACRO will be developed here. Thus, one might define

```
FADD    MACRO    A,B,C
        CLA      A
        FAD      B
        STO      C
        END
```

Then, the call

```
        FADD     (A,1),(B,2),(A,1)
```

produces the sequence

```
        CLA      A,1
        FAD      B,2
        STO      A,1
```

However, when one starts programming with this definition of 'FADD', one soon realizes a shortcoming. Quite often the first number is already in the accumulator (AC) or is in the multiplier-quotient (MQ), and hence the definition is no longer efficient. Thus, one might change the definition to

```
FADD    MACRO    A,B,C
        WHEN     /H/AC,/NS/.E.,/H/A
        SKIP     5
        WHEN     /H/MQ,/NS/.E.,/H/A
        SKIP     2
        CLA      A
        SKIP     1
        XCA
        FAD      B
        STO      C
        END
```

Thus, the call

```
        FADD     (A,1),(B,2),(A,1)
```

produces the sequence

```
        CLA      A,1
        FAD      B,2
        STO      A,1
```

while the call

```
        FADD     A,C,(B,2),(A,1)
```

produces the sequence

```
        FADD    B,2
        STO     A,1
```

and the call

```
        FADD    MQ,(B,2),(A,1)
```

produces the sequence

```
        XCA
        FAD     B,2
        STO     A,1
```

In a similar fashion, the 'FADD' definition may be extended so that the second and third arguments are treated in a like manner. An interesting MACRO definition to experiment with is one in which the symbols 'AC' and 'MQ' may be used, as above, in any combination in the three MACRO arguments.

B. 'CALL' and 'CALLIO' as MACROS.

Another use of MACROS is to define operations which simply do not exist in UMAP. Suppose, for example, that UMAP did not contain the 'CALL' and 'CALLIO' pseudo-ops. These could then be defined as MACROS. One such set of MACRO definitions is the following.

```
CALL    MACRO   NAME,ARG
        TSX     /TV/NAME,4
        IRP     ARG
        PARAM   TXH,TIX,ARG
        IRP
        END
CALLIO  MACRO   NAME,ARG
        TSX     /TV/NAME,4
        IRP     ARG
        PARAM   IOP,IOP,ARG
        IRP
        END
PARAM   MACRO   OP1,OP2,AR1,AR2,AR3
        IFF     1,/CRS/AR1
        SKIP
        IFF     1,AR2,...
        OP2     AR1,,AR3
        IFF     0,AR2,...
        OP1     AR1,,0
        END
```

Note that due to the 'IRP' pseudo-ops in the above definitions, certain parenthesizing is needed in calls on these MACROS which is not normally needed in using the 'CALL' and 'CALLIO' pseudo-ops. Thus, the call

```
        CALL    ZERO,((A,...,B),C,D)
```

results in the sequence

```
        TSX     /TV/ZERO,4
        TIX     A,,B
        TXH     C,,0
        TXH     D,,0
```

while the call

```
        CALLIO  .PRINT,(A,(C,...,C+10),=K1234,0)
```

results in the sequence

```
        TSX     /TV/.PRINT,4
        IOP     A,,0
        IOP     C,,C+10
        IOP     =K1234,,0
        IOP     0,,0
```

Further, given these definitions, one could easily define additional pseudo-ops such as 'READ' and 'PAUSE' as

```
READ    MACRO   ARG
        CALLIO  .READ,(ARG)
        END
PAUSE   MACRO   ARG
        CALL    .PAUSE,(ARG)
        END
```

C. Key Word Formation MACRO

Consider the following task. Suppose that a program reads, from data cards, words of length greater than 6 characters, so that these words cannot be stored in BCD form in single locations. To prevent the usage of excess machine storage, the programmer decides that he will recognize a given word, not by its total set of characters, but by looking at an abbreviated set of characters extracted from the given word. For example, he might decide to delete every other character of the word and keep only the last 6 of the remaining characters as a 'KEY WORD' with which to recognize the original word. Then, the program must contain a set of all possible 'KEY WORDS' in order to decipher according to this scheme. To help in writing the program, the programmer decides to build a set of Macros which will do this decoding for him at assembly time, thus saving him this effort. One such set is the following.

```
KEY     MACRO   A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S
        KEY1    0,0,0,0,0,B,D,F,H,J,L,N,P,R,S
        END
KEY1    MACRO   A,B,C,D,E,F,G,H,I,J,K,L,M,N
        IFF     0,/CRS/G,2
        BCI     1,B'B'C'D'E'F
        SKIP
        KEY1    B,C,D,E,F,G,H,I,J,K,L,M,N
        END
```

The MACRO 'KEY' deletes every other character of the original word (which is obviously supplied to it one character per argument), and the MACRO 'KEY1' shifts the remaining characters until the last 6 are obtained. It is assumed that the original word has at least 2 characters and at most 18 characters. Thus, the call

```
KEY     M,O,T,H,E,R
```

results in

```
BCI     1,0000HR
```

while the call

```
KEY     P,R,O,B,L,E,M,T,Y,P,E,S,Y,M,7,1
```

results in

```
BCI     1,ETPSM1
```

COMBINED OPERATIONS TABLE OF UMAP

The following table lists most of the machine operations, pseudo-operations, and extended machine operations which UMAP handles.

The TYPE column in the table contains up to 3 characters which help to describe the operations. The meaning of these characters are as follows.

I: The instruction is indirect addressable.
A: The instruction requires an address.
T: The instruction requires a tag.
D: The instruction requires a decrement.
4: The instruction is a 7094 instruction.
P: The instruction is a pseudo-instruction.
E: The instruction is an extended machine instruction.
M: The instruction is used mostly for MACRO definition.
N: The instruction is an indicator instruction.
B: The instruction requires a Boolean address.

All pseudo-operations have a page number enclosed in parenthesis. A description of the pseudo-operation may be found on the specified page.

All instructions not having P or E in the TYPE column are machine instructions and their descriptions may be found in the manual IBM 7094 Principles of Operations.

CODE	OCTAL CODE	TYPE	COMMENT AND/OR PAGE
9LP		P	(3.7-26) assembly control
ABS		P	(3.7-22) assembly control
ACL	0361	AI	add and carry logical word
ADD	0400	AI	add
ADM	0401	AI	add magnitude
ALS	0767	A	accumulator left shift
ANA	4320	AI	and to accumulator
ANS	0320	AI	and to storage
ARS	0771	A	accumulator right shift
AXC	4774	AT	address to index complemented
AXT	0774	AT	address to index true
BCD		P	(3.7-27) data generation
BCI		P	(3.7-27) data generation
BES		P	(3.7-27) storage allocation
BFT		EBN	(3.7-50) extended machine instruction
BLK	2000	E	(3.7-50) extended prefix code
BNT		EBN	(3.7-50) extended machine instruction
BRA	7000	EA	(3.7-50) extended prefix code
BSS		P	(3.7-28) storage allocation
BTS		P	(3.7-28) storage allocation
CAL	4500	AI	clear and add logical word
CAQ	4114	AD	convert by addition from MQ
CAS	0340	AI	compare AC with storage
CHS	0760...2		change sign of AC
CLA	0500	AI	clear and add
CLM	0760...0		clear magnitude of AC
CLS	0502	AI	clear and subtract
COM	0760...6		complement magnitude of AC
CRQ	4154	AD	convert by replacement from MQ
CVR	0114	AD	convert by replacement from AC
DCT	0760...12		divide check test
DEC		P	(3.7-29) data generation
DLD	0443	AI4	double load
DST	4603	AI4	double store
DVH	0220	AI	divide or halt
DVP	0221	AI	divide or proceed
ENB	0564	AI	enable channel interrupt
END		P	(3.7-32) assembly control
ENK	0760...4		enter keys from console to MQ
EQU		P	(3.7-33) symbol definition
ERA	0322	AI	exclusive or to accumulator
ETC		P	(3.7-34) statement continuation
ETM	0760...7		enter transfer trap mode
FAD	0300	AI	floating point add
FAM	0304	AI	floating point add magnitude
FDH	0240	AI	floating point divide or halt
FDP	0241	AI	floating point divide or proceed
FMP	0260	AI	floating point multiply
FMT	5000	EA	(3.7-50) extended prefix code
FOR	4000	E	(3.7-54) extended prefix code
FRN	0760...11		floating point round

CODE	OCTAL CODE	TYPE	COMMENT AND/OR PAGE
FSB	0302	AI	floating point subtract
FSM	0306	AI	floating point subtract magnitude
FUL		P	(3.7-35) assembly control
FVE	5000	E	(3.7-54) extended prefix code
HPR	0420	A	halt and proceed
HTR	0000	AI	halt and transfer
IFF		PM	(3.7-67) MACRO control
IIA	0041	N	invert indicators from AC
IIB		EBN	(3.7-50) extended machine instruction
IIL	4051	BN	invert indicators of left half
IIR	0051	BN	invert indicators of right half
IIS	0440	BIN	invert indicators from storage
IOP	5000	EA	(3.7-50) extended prefix code
IOT	0760...5		input/output check test
IRP		PM	(3.7-64) indefinite repeat
...	0000	E	(3.7-54) extended prefix code
LAC	0535	AT	load complement of address in index
LAS	4340	AI	logical compare AC with storage
LBL		P	(3.7-53) assembly control
LBT	0760...1		low bit test
LDC	4535	AT	load complement of decrement in index
LDI	0441	AIN	load indicators from storage
LDQ	0560	AI	load MQ
LFT	4054	BN	left half indicators off test
LGL	4763	A	logical left shift
LGR	4765	A	logical right shift
LLS	0763	A	long left shift
LNT	4056	BN	left half indicators on test
LOC		P	(3.7-36) assembly control
LRS	0765	A	long right shift
LTM	4760...7		leave trapping mode
LXA	0534	AT	load index from address
LXD	4534	AT	load index from decrement
MON	5000	E	(3.7-54) extended prefix code
MPR	4200	AI	multiply and round
MPY	0200	AI	multiply
MSE	4760	A	minus sense
MTH	7000	E	(3.7-54) extended prefix code
MTW	6000	E	(3.7-54) extended prefix code
MZE	4000	E	(3.7-54) extended prefix code
NOP	0761		no operation
NZT	4520	AI	storage non-zero test
OAI	0043	N	logical or AC to indicators
OCT		P	(3.7-37) data generation
OFT	0444	AIN	off test for indicators
ONE	1000	E	(3.7-54) extended prefix code
ONT	0446	AIN	on test for indicators
OPD		P	(3.7-54) operation definition
ORA	4501	AI	logical or to accumulator
ORG		P	(3.7-38) assembly control
ORS	4602	AI	logical or to storage
OSI	0442	AIN	logical or storage to indicators

CODE	OCTAL CODE	TYPE	COMMENT AND/OR PAGE
PAC	0737	T	place complement of address in index
PAI	0044	N	place accumulator in indicator
PAR	3000	EA	(3.7-51) extended prefix code
PAX	0734	T	place address in index
PBT	4760...1		P-bit test
PCA	0756	T4	place complement of index in address
PCC		P	(3.7-39) assembly list control
PCD	4756	T4	place complement of index in decrement
PDC	4737	T	place complement of decrement in index
PDX	4734	T	place decrement in index
PGM		P	(3.7-40) assembly control
PIA	4046	N	place indicators in accumulator
PMC		PM	(3.7-84) assembly list control
PON	1000	E	(3.7-54) extended prefix code
PSE	0760	A	plus sense
PST		P	(3.7-41) assembly control
PTH	3000	E	(3.7-54) extended prefix code
PTW	2000	E	(3.7-54) extended prefix code
PXA	0754	T	place index in address
PXD	4754	T	place index in decrement
PZE	0000	E	(3.7-54) extended prefix code
RCT	0760...14		restore channel traps
RDS	0762	A	read select
REF		P	(3.7-41) assembly list control
REL		P	(3.7-42) assembly control
REM		P	(3.7-42) remark pseudo-op
RFT	0054	BN	right half indicators off test
RIA	4042	N	reset indicators from accumulator
RIB		EBN	(3.7-51) extended machine instruction
RIL	4057	BN	reset indicators of left half
RIR	0057	BN	reset indicators of right half
RIS	0445	AIN	reset indicators from storage
RMT		PM	(3.7-43) assembly control
RND	0760...10		round
RNT	0056	BN	right half indicators on test
RQL	4773	A	rotate MQ left
RST		P	(3.7-43) assembly control
***	0000	E	(3.7-54) extended prefix code
SAK		P	(3.7-43) assembly control
SBM	4400	AI	subtract magnitude
SCA	0636	AT4	store complement of index in address
SCD	4636	AT4	store complement of index in decrement
SET		P	(3.7-44) assembly control
SIB		EBN	(3.7-51) extended machine instruction
SIL	4055	BN	set indicators of left half
SIR	0055	BN	set indicators of right half
SIX	6000	E	(3.7-54) extended prefix code
SLF	0760...140		(3.7-51) sense lights off
SLN	0760...14X	A	(3.7-51) sense light on
SLQ	4620	AI	store left half of MQ
SLT	4760...14X	A	(3.7-51) sense light test

CODE	OCTAL CODE	TYPE	COMMENT AND/OR PAGE
SLW	0602	AI	store logical word
SSM	4760...3		set sign minus
SSP	0760...3		set sign plus
SST		P	(3.7-44) symbol definition
STA	0621	AI	store address
STD	0622	AI	store decrement
STI	0604	AIN	store indicators
STL	4625	AI	store location counter
STO	0601	AI	store
STP	0630	AI	store prefix
STQ	4600	AI	store MQ
STR	5000		store location and trap
STT	0625	AI	store tag
STZ	0600	AI	store zero
SUB	0402	AI	subtract
SVN	7000	E	(3.7-54) extended prefix code
SWT	0760...16X	A	(3.7-51) sense switch test
SXA	0634	AT	store index in address
SXD	4634	AT	store index in address
SYN		P	(3.7-45) symbol definition
TCD		P	(3.7-46) assembly control
TCH	1000	A	transfer in channel
TIF	0046	AIN	transfer if indicators off
TIO	0042	AIN	transfer if indicators on
TIX	2000	ATD	transfer on index
TLQ	0040	AI	transfer on low MQ
TMI	4120	AI	transfer on minus AC
TNO	4140	AI	transfer on no overflow
TNX	6000	ATD	transfer on no index
TNZ	4100	AI	transfer if AC not zero
TOV	0140	AI	transfer on overflow
TPL	0120	AI	transfer if AC plus
TQO	0161	AI	transfer on MQ overflow
TQP	0162	AI	transfer on MQ plus
TRA	0020	AI	transfer
TSX	0074	AT	transfer and set index
TTL		P	(3.7-53) assembly list control
TTR	0021	AI	trap transfer
TWO	2000	E	(3.7-54) extended prefix code
TXH	3000	ATD	transfer on index high
TXI	1000	ATD	transfer with index incremented
TXL	7000	ATD	transfer on index low or equal
TZE	0100	AI	transfer if AC zero
UAM	4304	AI	unnormalized add magnitude
UFA	4300	AI	unnormalized floating add
UFM	4260	AI	unnormalized floating multiply
UFS	4302	AI	unnormalized floating subtract
USM	4306	AI	unnormalized subtract magnitude
VDH	0224	ATD	variable length divide or halt
VDP	0225	ATD	variable length divide or proceed
VFD		P	(3.7-46) data generation

CODE	OCTAL CODE	TYPE	COMMENT AND/OR PAGE
VLM	0204	ATD	variable length multiply
WRS	0766	A	write select
XCA	0131		exchange AC and MQ
XCL	4130		exchange logical AC and MQ
XEC	0522	AI	execute instruction
ZAC	4754	EA	(3.7-52) extended machine instruction
ZAD	0634	EA	(3.7-52) extended machine instruction
ZDC	4634	EA	(3.7-52) extended machine instruction
ZET	0520	AI	zero storage test
ZMT		PM	(3.7-85) MACRO control
ZSA	0634	EA	(3.7-52) extended machine instruction
ZSD	4634	EA	(3.7-52) extended machine instruction
ZST		P	(3.7-49) assembly control
BOOL		P	(3.7-28) symbol definition
CALL		P	(3.7-28) subroutine call
DATE		P	(3.7-29) assemble current data
DFAD	0301	AI4	double precision add
DFAM	0305	AI4	double precision add magnitude
DFDH	4240	AI4	double precision divide or halt
DFDP	4241	AI4	double precision divide or proceed
DFMP	0261	AI4	double precision multiply
DFSB	0303	AI4	double precision subtract
DFSM	0307	AI4	double precision subtract magnitude
DUAM	4305	AI4	double precision unnorm. add magnitude
DUFA	4301	AI4	double precision unnorm. add
DUFM	4261	AI4	double precision unnorm. multiply
DUFS	4303	AI4	double precision unnorm. subtract
DUSM	4307	AI4	double precision unnorm. subtract magnitude
EFTM	4760...2		enter floating trap mode
EMTM	4760...16		enter multiple tag mode
ERAS		P	(3.7-33) storage allocation
EVEN		P	(3.7-52) storage allocation
FIVE	5000	E	(3.7-54) extended prefix code
FOUR	4000	E	(3.7-54) extended prefix code
HEAD		P	(3.7-35) assembly control
IOBP	4000	EAD	(3.7-50) extended I/O command
IOBT	3000	EAD	(3.7-50) extended I/O command
IOCD	0000	AD	I/O under count and disconnect
IOCP	4000	AD	I/O under count and proceed
IOCT	5000	AD	I/O under count and transfer
IORP	2000	AD	I/O a record and proceed
IORT	3000	AD	I/O a record and transfer
IOSP	6000	AD	I/O until signal and proceed
IOST	7000	AD	I/O until signal and transfer
LFTM	4760...4		leave floating trap mode
LMTM	0760...16		leave multiple tag mode
LIST		P	(3.7-36) assembly control
LOOK		P	(3.7-36) .LOOK call
NULL		P	(3.7-37) assembly control
READ		P	(3.7-41) .PRINT call
SAVE		P	(3.7-43) .SAVE call

CODE	OCTAL CODE	TYPE	COMMENT AND/OR PAGE
SKIP		PM	(3.7-66) MACRO control
WHEN		PM	(3.7-76) MACRO control
ZERO		P	(3.7-49) define cleared cells
BRIEF		P	(3.7-28) assembly list control
COUNT		P	(3.7-53) assembly control
EJECT		P	(3.7-32) assembly list control
ENDIO		P	(3.7-19) I/O pseudo-op
ENTRY		P	(3.7-33) defines entry point
EXECT		P	(3.7-34) binary output control
INDEX		P	(3.7-36) assembly list control
IOBPN	4000	EAD	(3.7-50) extended I/O command
IOBTN	3000	EAD	(3.7-50) extended I/O command
IOCDN	0000	AD	non-transmit I/O and disconnect
IOCPN	4000	AD	non-transmit I/O and proceed
IOCTN	5000	AD	non-transmit I/O and transfer
IORPN	2000	AD	non-transmit I/O and proceed
IORTN	3000	AD	non-transmit I/O and transfer
IOSPN	6000	AD	non-transmit I/O and proceed
OPSTN	7000	AD	non-transmit I/O and transfer
IOTRA	1000	EA	(3.7-51) used in calling low core I/O
MACRO		PM	(3.7-55) MACRO definition
NEWID		P	(3.7-36) assembly punch control
NOCOM		P	(3.7-37) assembly list control
NOCRS		PM	(3.7-81) MACRO control
NOMAC		PM	(3.7-84) MACRO list control
NONOP		P	(3.7-37) operation definition
OBJCT		P	(3.7-37) assembly punch control
OPSYN		P	(3.7-38) operation definition
PAUSE		P	(3.7-39) .PAUSE call
PCCON		P	(3.7-39) assembly list control
PMCON		PM	(3.7-84) MACRO list control
PRINT		P	(3.7-41) .PRINT call
PUNCH		P	(3.7-41) .PUNCH call
REFON		P	(3.7-42) assembly control
SETTO		P	(3.7-44) .SET call
SEVEN	7000	E	(3.7-54) extended prefix code
SPACE		P	(3.7-44) assembly list control
START		P	(3.7-44) define entry point
THREE	3000	E	(3.7-54) extended prefix code
TITLE		P	(3.7-46) assembly list control
ASSIGN		P	(3.7-26) storage allocation
BINARY		P	(3.7-27) assembly control
BRANCH	7000	EA	(3.7-50) extended operation
CALLIO		P	(3.7-29) I/O subroutine call
COMMNT		P	(3.7-29) .COMNT call
COMMON		P	(3.7-52) storage allocation
DECMOD		P	(3.7-32) set decimal mode
DESIST		P	(3.7-53) stop assembly process
DETAIL		P	(3.7-32) assembly list control
ENDPGM		P	(3.7-32) assembly control
EQUMAX		P	(3.7-33) symbol definition

CODE	OCTAL CODE	TYPE	COMMENT AND/OR PAGE
EQUMIN		P	(3.7-33) symbol definition
ERLIST		P	(3.7-34) storage allocation
EXTERN		P	(3.7-34) transfer vector control
FLAGOP		P	(3.7-34) assembly list control
FLAGSY		P	(3.7-34) assembly list control
MIDDLE		P	(3.7-36) assembly control
NEXECT		P	(3.7-37) assembly punch control
NOBJCT		P	(3.7-37) assembly punch control
NOLIST		P	(3.7-37) assembly list control
OCTMOD		P	(3.7-38) assembly control
ORGCRS		PM	(3.7-83) MACRO control
PCCOFF		P	(3.7-39) assembly list control
PCLIST		P	(3.7-39) assembly list control
PCMORG		P	(3.7-39) assembly control
PGMCOM		P	(3.7-40) storage allocation
PMCOFF		PM	(3.7-84) MACRO list control
PUNLIT		P	(3.7-41) assembly list control
REFOFF		P	(3.7-41) assembly list control
RELIST		P	(3.7-42) assembly list control
RESERS		P	(3.7-42) assembly control
RESPGC		P	(3.7-42) assembly control
RESTOR		P	(3.7-42) .RSTOR call
RESUME		P	(3.7-53) resume assembly process
SYMBOL		P	(3.7-45) adjust symbol table length
TAPENR	5000	EA	(3.7-52) extended operation
TAPERD		P	(3.7-45) .TAPRD call
TAPEWR		P	(3.7-45) .TAPWR call
UNLIST		P	(3.7-37) assembly list control

FAP Operations Not In UMAP

The following machine operations available in the FAP Assembly Program are not defined in UMAP. They may, if necessary, be defined in a program through use of the 'OPD' pseudo-operation.

BSFX	SCHX	WTBX
BSRX	SDNX	WTDX
BTTX	SDLX	ECTM
ETTX	SPRX	ESNT
LCHX	SPTX	LSNM
RCDX	SPUX	
RCHX	STCX	
RDCX	TCNX	
REWX	TCOX	
RICX	TEFX	
RPRX	TRCX	
RSCX	WEFX	
RTBX	WPBX	
RTDX	WPDX	
RUNX	WPUX	

Where X = A,B,C,D,E,F,G, or H.

Also, 7909 data channel commands, 7631 file control orders, and 7640 hypertape control orders are not defined in UMAP.

FAP Pseudo-Operations Not In UMAP

The pseudo-operations TAPENO, DUP, IFEOF, 704, 7090, OPVFD, MOP and MAC which are available in FAP are not defined for UMAP.

Also, the $ notation for subroutine calls is not available in UMAP.

FAP Pseudo-Operations Which Operate Differently In UMAP

The FAP pseudo-operations HEAD, COUNT, TITLE, SST and TTL are defined differently from the same pseudo-operations in UMAP.

Use of the FAP pseudo-operations COUNT, TITLE, and TTL in a UMAP program will not affect the correctness of the assembly.

MAMOS LIBRARY SUBROUTINES

INTRODUCTION

PURPOSE: This section of the MAMOS manual is designed to make the user of MAMOS aware of the subroutines available to him, and to give him the necessary information concerning the use of those subroutines. Not all subroutines available to the user are listed in this collection. The low-core subroutines are described elsewhere in the manual, and are not included in this section.

The user will find, at the end of this section, an index of the subroutines, both by name and by function. Included in the index are the entry points to the subroutines, the length of the subroutines in octal and the number of erasable locations used in octal. The length is given as a means of calculating the amount of core space occupied by a program and its subroutines.

CALLING SEQUENCES: Calling sequences will be given in those languages (i.e., UMAP, MAD or FORTRAN) for which the subroutine is designed to work. The name of the subroutine may change depending on the language used in calling it. In writing FORTRAN programs to be translated by MADTRAN, the user must remember that the source program given to the compiler is actually in MAD. The decision as to whether to use the FORTRAN or MAD calling sequence for subroutine calls will depend upon the particular function of the entry point in question, and is left up to the user. In general, if a calling sequence is not given for a particular language, the routine cannot be used in that language.

ARGUMENTS: The descriptions of the arguments in subroutine write-ups are usually a source of confusion. To aid the user, certain terms and phrases will be defined here. It is important that the distinctions made here be remembered when consulting any of the write-ups in this section.

1. In MAD and FORTRAN, when an argument is a single variable (not an array), the letter specifying the argument stands for a constant, the name of a variable, or (unless otherwise specified) an expression. The value of each argument is the value of the variable in the mode specified. In UMAP, the letter is the symbolic address of the value of the variable.
2. Array designations may occur in two ways, either by the name of the variable or by its first entry (or, equivalently, its first element or base element). Throughout this section, the word array will be used to indicate either a vector or a higher dimension array.
   A. ARRAY NAME: If the data is stored in array A, then the name of the array is A. In MAD, the subroutine will locate the dimension vector of the array and use this information to find any other element of the array. In FORTRAN, the name A is equivalent to the base element of the array. In UMAP, the name A refers to the symbolic address of the first entry.
   B. FIRST ENTRY OF ARRAY: If the data stored in the array begins in A(8), then A(8) must be given in the calling sequence if the first entry is required. Note that A(1,1) is usually the first entry of a two-dimension array. In UMAP, the first entry is the symbolic address of the first data element.

INTRODUCTION (CONTINUED)

3. Many subroutines have a computation switch, usually of floating point mode. (This mode is used due to the difference between MAD and FORTRAN integers.) Therefore, one may write:

```
MAD      I = SUB.(ARGS)
FORTRAN I = SUB (ARGS)
```

I will automatically be declared an integer in FORTRAN and must be so declared in MAD. The translators will convert to the proper form of the integer. Further, in MAD one may write:

```
TRANSFER TO S (SUB.(ARGS))
```

where S(I) is a statement label. Here again, the integer conversion is automatically provided.

4. If a subroutine uses N cells of erasable storage, then they will be the high N locations in core storage. Thus, if 3 erasable locations are used, they will be -1 = 77777, -2 = 77776, and -3 = 77775.
5. Many of the subroutines store their arguments in erasable. It is of great aid when debugging programs (with the aid of a dump), to have this information available. For this reason, often the erasable location where the argument is stored is noted, enclosed in parentheses, after the description of the argument itself.
6. In UMAP, some of the arguments may be tagged and the tag will be effective in computing the effective address of the argument. For instance:

```
CALL SIN
PAR ARG,2
```

will compute the sine of the value found in location (ARG-C(IR2)). Naturally, Index register 4 cannot be used for this purpose. Unless specifically mentioned, arguments cannot be tagged. In the following write-ups, these tags will be referred to as optional tags.

7. Some subroutines require as arguments, a location to which control may be transferred by the subroutine. These can be given directly by MAD or by UMAP, but to do this in FORTRAN requires the following dodge. Assume that we wish to give the subroutine named XXX the formula number 6 as an argument. A possible sequence is then:

```
      ASSIGN 60 to N
      GO TO N,(6,60)
   60 ASSIGN 6 to N
      CALL XXX (N)
```

Or, another possibility is:

```
      ASSIGN 6 to N
      IF (I) 3, 3, 4
   3  GO TO N, (6)
   4  CALL XXX (N)
```

The main point is that the N must be used in at least one assign statement and N must be tested at least once by an assigned GO TO.

NOTES:

1. In MAD there exists a special subscripting option (see the MAD manual), which allows a user to define his own method of computing subscripts. Those arrays which utilize this option may not be used as input arguments to subroutines not written in MAD (which includes almost all library subroutines), which require as input the name or first element of an array. All arrays used as input to the library routines must be dimensioned in the standard way.

INTRODUCTION (CONTINUED)

2. There are several equivalent ways to call a subroutine using UMAP. In any given write-up, only one way will be given, since the others are directly derivable. In general, if XXX is the name of a subroutine, L is a single parameter and the block B to B+10 is another parameter:

```
       CALL XXX,L,B,...,B+10
```

   is equivalent to

```
       CALL XXX
       PAR  L
       BLK  B,,B+10
```

   is equivalent to

```
       TSX  /TV/XXX,4  (The /TV/ is needed only if UMAP is
       PAR  L           generating the program card.)
       BLK  B,,B+10
```

   See the UMAP write-up in this manual for further details.

3. All library subroutines follow the share conventions and preserve index registers and sense indicators. However, unless it is otherwise stated, the subroutines will not necessarily preserve the AC, MQ, overflow light, divide check light or sense lights.

The majority of this section of the MAMOS manual comes from the MESS write-up of the University of Michigan Executive System.

The following symbols are used in this write-up for describing limits, ranges and magnitudes of arguments and results:

SYMBOL	MEANING
.E.	Equal to.
.NE.	Not equal to.
.P.	To the power.
.G.	Greater than.
.GE.	Greater than or equal to.
.L.	Less than.
.LE.	Less than or equal to.
PI	π

LOGICAL OPERATIONS

ENTRY POINTS: ANA, ORA

PURPOSE: Provide the logical operations AND and OR for use in MAD programs.

CALLING SEQUENCES:

```
MAD       X = ANA.(A,B)      (AND)
MAD       X = ORA.(A,B)      (OR)
```

ARGUMENTS:

```
A         First argument
B         Second argument
X         Bitwise OR (ORA.) or bitwise AND (ANA.) of the 36-bit
          arguments A and B.
```

FLOATING-POINT ARCSINE AND ARCCOSINE

ENTRY POINTS: ARCSIN, ARCCOS

PURPOSE: Compute ARCSIN(X) and ARCCOS(X) for floating-point argument X.

CALLING SEQUENCES:

```
MAD       Y = ARCSIN.(X)
          Y = ARCCOS.(X)
FORTRAN   Y = ARCSIN(X)
          Y = ARCCOS(X)
UMAP      CALL ARCSIN
          PAR X,T
          NORMAL RETURN - RESULT IN AC.

          CALL ARCCOS
          PAR X,T
          NORMAL RETURN - RESULT IN AC.
```

ARGUMENTS:

```
X         Floating-point argument for which the arcsine or arccosine
          is desired.  X must satisfy the inequality .ABS.X .LE. +1.
Y         The resulting functional value.
               For the arcsin routine, Y will fall in the interval
               -PI/2.LE.Y.LE.+PI/2  .
               For the arccos routine, Y will fall in the interval
               0.LE.Y.LE.PI  .
T         Optional tag.
```

SUBROUTINES REQUIRED: SQRT

ERROR CONDITION:

.ABS.(X) .G. +1
This causes an argument to the SQRT routine to be negative.
SQRT will print out a comment to this effect, and control will be returned to the system error routine.

FLOATING POINT PRINCIPLE VALUED ARCTANGENT

ENTRY POINTS: ATAN

PURPOSE: Compute principle value of ARCTAN(X) for floating point argument X.

CALLING SEQUENCES:

```
    MAD       Y = ATAN.(X)
    FORTRAN   Y = ATAN(X)
    UMAP      CALL   ATAN
              PAR    X,T
              NORMAL RETURN - Y IN THE ACCUMULATOR.
```

ARGUMENTS:

X	The floating point argument for which the arctangent is desired.
Y	The desired angle in floating point, -PI/2.LE.Y.LE.PI/2.
T	Optional tag.

OCTAL LOCATION FINDER

ENTRY POINTS: ATLOC

PURPOSE: Prints ' AT LOCATION XXXXX', where XXXXX is either a location given or the location ATLOC was called from.

CALLING SEQUENCES:

```
    MAD       EXECUTE ATLOC.(LOC)
    UMAP      CALL ATLOC,LOC
    FORTRAN   CALL ATLOC(LOC)
```

ARGUMENTS:

LOC	When non-zero, the 2's complement of LOC will be printed. When zero, the location from which ATLOC is called will be printed.

SUBROUTINES REQUIRED: SPRINT

FLOATING POINT SINGLE VALUED ARCTANGENT

ENTRY POINTS: ATN1

PURPOSE: Compute in the range of 0 to 2PI the single value of ARCTAN(Y/X) for floating point arguments X and Y.

CALLING SEQUENCES:

```
    MAD       Z = ATN1.(Y,X)
    FORTRAN   Z = ATN1(Y,X)
    UMAP      CALL   ATN1
              PAR    Y,T1
              PAR    X,T2
              NORMAL RETURN - Z IN THE ACCUMULATOR.
```

(FLOATING POINT SINGLE VALUED ARCTANGENT - CONTINUED)

ARGUMENTS:

X,Y	The floating arguments, where $TAN^{-1}(Y/X)$ is the desired angle.
Z	The desired angle in floating point (radians), 0.LE.Z.LE.2PI.
T1,T2	Optional tags.

SIMULTANEOUS LINEAR EQUATIONS

ENTRY POINTS: BAKSUB

PURPOSE: Perform the double back-substitution L * Y = B and R * Z = Y where L is a monic lower triangular matrix and R is upper triangular. The sub-diagonal elements of L and the non-zero elements of R are assumed to be stored in the same matrix A (see subroutine GAUSS).

RESTRICTION: No check for singularity or inconsistency is made by this subroutine. It is, therefore, advised that the solution be checked. All checking is left up to the user.

CALLING SEQUENCES:

```
FORTRAN  X = BAKSUB ( N,A,Z,B )
MAD      X = BAKSUB.( N,A,Z,B )
UMAP     CALL   BAKSUB,N,A,Z,B
         RETURN - FLOATING-POINT SWITCH IN AC.
```

ARGUMENTS:

N	Integer dimension of the square matrix A.
A	First element of the matrix A in which L and R are stored. For further information see the write-up for subroutine GJRDT.
Z	For a successful return, Z will be the solution vector.
B	Right hand side of the system of equations.
X	Floating-point switch 1. Successful computation. 0. Overflow, cannot continue.

SUBROUTINES REQUIRED: FSPILL,RSPILL

ONE WORD BCD TO BINARY CONVERSION

ENTRY POINTS: BCDBN, MBCDBN

PURPOSE: Convert one BCD word into the equivalent integer.

CALLING SEQUENCES:

```
MAD       I = BCDBN.(N)
FORTRAN   I = MBCDBN(N)
UMAP      CALL    BCDBN
          PAR     N,T
          NORMAL RETURN - I IN THE ACCUMULATOR
```

ARGUMENTS:

N	The BCD word to be converted to an integer. All blanks are completely ignored.
I	The integer equivalent to the BCD argument, N.
T	Optional tag.

ONE WORD BINARY TO BCD CONVERSION

ENTRY POINTS: BNBCD

PURPOSE: Convert a binary integer into its BCD equivalent which is right justified and filled in with leading zeros. The primary use of this subroutine is modifying formats with integers read in as data.

CALLING SEQUENCES:

```
MAD       N = BNBCD.(I)
FORTRAN   X = BNBCD(I)
UMAP      CALL    BNBCD
          PAR     I,T
          NORMAL RETURN - N IN THE ACCUMULATOR.
```

ARGUMENTS:

I	The integer to be converted.
N	The BCD equivalent of the integer I. Zeros are filled in on the left to complete the BCD word.
T	Optional tag.

MATRIX INVERSION

ENTRY POINTS: BORDS

PURPOSE: Inversion of a real symmetric matrix upon itself. The determinant is calculated as a by-product. The matrix is inverted by the method of successive bordering. For success, this method requires that submatrices of the form

```
A(1,1) ... A(1,J)
  .  .  .  .
A(J,1) ... A(J,J)
```

for J = 1,2,...,N be non-singular. If the determinant of the matrix is non-zero, this condition is always satisfied.

CALLING SEQUENCES:

FORTRAN	See write-up for subroutine IBDS.
MAD	X = BORDS. (N,A,D)
UMAP	CALL BORDS,N,A,D RETURN - X WILL BE IN THE AC

ARGUMENTS:

N	Integer dimension of the square matrix A. N may not exceed 100.
A	First element of the matrix. For further information, see the write-up for the subroutine GJRDT.
D	After a successful return, D will contain the determinant.
X	Floating-point switch; 1. Successful inversion. -K. Negative integer giving the dimension of the smallest principal minor equal to zero.

BESSEL FUNCTIONS

ENTRY POINTS: BSL1

PURPOSE: Compute Bessel functions J(N,X), I(N,X), Y(N,X), K(N,X), EXP(-X)*I(N,X), and EXP(X)*K(N,X), for real values of X.GE.0. and for integral values of N, 0.LE.N.LE.5, where N is the order of the desired Bessel function.

CALLING SEQUENCES:

```
MAD       L = BSL1.(X,I,N,B,K)
FORTRAN   L = BSL1(X,I,N,B,K)
UMAP      CALL    BSL1
          PAR     X
          PAR     I
          PAR     N
          PAR     B
          PAR     K
          NORMAL RETURN - L IN THE ACCUMULATOR.
```

ARGUMENTS:

X — Floating point argument X.

I — Integer argument specifying the type of Bessel function desired.
- 1 for J(N,X)
- 2 for I(N,X)
- 3 for K(N,X)
- 4 for Y(N,X)
- 5 for EXP(-X)*I(N,X)
- 6 for EXP(X)*K(N,X)

N — Integer argument N, the order of the desired function, 0.LE.N.LE.5.

B — The desired Bessel function of the argument X.

K — Integer argument which indicates the number of binary digits desired for convergence. K = 0, then all bits must agree, K = 1, then 26 digits are asked for, K = 5 specifies 22 bits, etc.

L — Computation flag (floating point).
- L = 1. Successful return. Desired Bessel function stored in B.
- L = 2. Error return. Specified arguments would result in function too large for machine.

SUBROUTINES REQUIRED: SQRT, COS, SIN, EXP, ELOG

MATRIX FACTORIZATION BY CHOLESKY DECOMPOSITION

ENTRY POINTS: CHOLES

PURPOSE: Factorization of a real symmetric positive definite matrix A into an upper triangular matrix R and a lower triangular matrix L such that

L = (R TRANSPOSE) and A = L * R.

Only the diagonal and upper diagonal elements need be stored in A on entry, and on exit the matrix L will be stored in the diagonal and lower diagonal elements. The above diagonal elements of A are not changed by CHOLES. If the matrix is not positive definite, this symmetric factorization is impossible without introducing imaginary elements in the factors.

CALLING SEQUENCES:

FORTRAN	X = CHOLES (N,M,A)
MAD	X = CHOLES. (N,M,A)
UMAP	CALL CHOLES,N,M,A
	RETURN - X WILL BE IN THE AC.

ARGUMENTS:

N	Integer dimension of the square matrix A.
M	Integer row length of matrix as stored in core storage.
A	First element of the matrix. For further information, see the write-up for the subroutine GJRDT.
X	Floating-point switch.
	1. Successful decomposition.
	0. Overflow, cannot continue.
	-1. Cholesky decomposition impossible. See above. The smallest principal minor that is non-positive definite may be found by examining the diagonal elements of the matrix on return. If A(I,I) is zero or negative, the I-th order minor is non-positive definite.

SUBROUTINES REQUIRED: FSPILL, NASQ1, RSPILL

COMPLEX ARITHMETIC

ENTRY POINTS: CMADD, CMSUB, CMMUL, CMDIV

PURPOSE: Does floating point complex arithmetic.

CALLING SEQUENCES:

MAD	EXECUTE CMADD. (R1,I1,R2,I2,RANS,IANS)	Addition
	EXECUTE CMSUB. (R1,I1,R2,I2,RANS,IANS)	Subtraction
	EXECUTE CMMUL. (R1,I1,R2,I2,RANS,IANS)	Multiplication
	EXECUTE CMDIV. (R1,I1,R2,I2,RANS,IANS,ERR)	Division
FORTRAN	CALL CMADD (R1,U1,R2,U2,RANS,UANS)	Addition
	CALL CMSUB (R1,U1,R2,U2,RANS,UANS)	Subtraction
	CALL CMMUL (R1,U1,R2,U2,RANS,UANS)	Multiplication
	CALL CMDIV (R1,U1,R2,U2,RANS,UANS,ERR)	Division

UMAP

Addition	Subtraction	Multiplication	Division
CALL CMADD	CALL CMSUB	CALL CMMUL	CALL CMDIV
PAR R1	PAR R1	PAR R1	PAR R1
PAR I1	PAR I1	PAR I1	PAR I1
PAR R2	PAR R2	PAR R2	PAR R2
PAR I2	PAR I2	PAR I2	PAR I2
PAR RANS	PAR RANS	PAR RANS	PAR RANS
PAR IANS	PAR IANS	PAR IANS	PAR IANS
			PAR ERR

ARGUMENTS:

R1	Real part of first operand.
I1, U1	Imaginary part of first operand.
R2	Real part of second operand.
I2, U2	Imaginary part of second operand.
RANS	Real part of answer.
IANS, UANS	Imaginary part of answer.
ERR	Use of this argument is optional. If used, attempted division by zero will cause a return to the location specified. Otherwise, execution will be terminated. (Fortran users should see point 7 under notes in the introduction to this section.)

ERROR CONDITION:

If R2 and I2 both equal zero when using CMDIV and ERR is given, control will be transferred to the location specified. Otherwise, the comment '**** COMPLEX DIVISION BY ZERO' will be printed and a dump given if requested.

SUBROUTINE REQUIRED: .EXIT FOR CMDIV

COMPLEX SQUARE ROOT

ENTRY POINTS: CMSQRT

PURPOSE: Compute square root of complex number.

CALLING SEQUENCES:

```
MAD     EXECUTE CMSQRT.(A,B,C,D,)
UMAP    CALL    CMSQRT
        PAR     A
        PAR     B
        PAR     C
        PAR     D
        NORMAL RETURN WITH REAL PART IN AC,  IMAGINARY PART IN MQ.
```

ARGUMENTS:

A	Real part of argument.
B	Imaginary part of argument.
C	Real part of result.
D	Imaginary part of result.

SUBROUTINES REQUIRED: SQRT

SYMBOL MANIPULATION - PACKING

ENTRY POINTS: COMPZ, ZCOMPZ

PURPOSE: COMPZ. Packs a sequence of left-justified BCD characters into full word (6 characters per word) BCD form. If the last word is incomplete, it is left-justified with trailing blanks. ZCOMPZ. is the same as COMPZ., except that the last word, if incomplete, is right-justified with leading zeroes.

CALLING SEQUENCES:

```
MAD     X = COMPZ.(M,A)
MAD     EXECUTE COMPZ.(M,A,B)
MAD     X = ZCOMPZ.(M,A)
MAD     EXECUTE ZCOMPZ.(M,A,B)
```

ARGUMENTS:

A	First entry of a vector of left-justified BCD characters which are to be packed into full-word form.
B	If B is given, the results of the packing process are stored in sequence, starting in B. If B is not given, (i.e., only M and A are given), only the first M, up to and including six, characters from A are packed.
M	Integer number of left-justified BCD characters to be used, starting in A.
X	Result of the packing process in the specific case only in which B is not given in the calling sequence.

MATRIX FACTORIZATION BY L-R DECOMPOSITION

ENTRY POINTS: CROUT

PURPOSE: Factorization of an arbitrary real matrix A into an upper triangular matrix R and a monic lower triangular matrix L. The original matrix is overwritten by the factors R and L in the same fashion as the subroutine GAUSS. Interchanges are not used and hence it may not be possible to factor the matrix. It should be kept in mind that this factorization without interchanges is an unstable numerical procedure, except for a small class of matrices. The best example of this class is the set of all real positive definite matrices. The factorization is accomplished by the compact form of Gaussian Elimination called the Crout reduction. The reader is referred to 'Introduction To Numerical Analysis' by Hildebrand, p. 429.

CALLING SEQUENCES:

```
    FORTRAN  X = CROUT  ( N,A )
    MAD      X = CROUT. ( N,A )
    UMAP     CALL    CROUT,N,A
             RETURN - X WILL BE IN THE AC.
```

ARGUMENTS:

N	Integer dimension of the square matrix A.
A	First element of the matrix. For further information, see the write-up for the subroutine GJRDT.
X	Floating-point switch.
	1 Successful decomposition.
	0. Overflow, cannot continue.
	-1. Factorization is impossible without interchanges.

SUBROUTINES REQUIRED: FSPILL, RSPILL

MATRIX FACTORIZATION BY L-R DECOMPOSITION

ENTRY POINTS: CROUTP

PURPOSE: Crout reduction with interchanges of an arbitrary real matrix A into two factors, an upper triangular matrix R and a monic lower triangular matrix L. The Crout reduction is a compact form of Gaussian Elimination and requires more running time but usually suffers from slightly less round-off error. The operation of CROUTP and GAUSS are otherwise the same. See subroutine GAUSS.

CALLING SEQUENCES:

FORTRAN	X = CROUTP (N,A,XCH)
MAD	X = CROUTP. (N,A,XCH)
UMAP	CALL CROUTP,N,A,XCH
	RETURN - X WILL BE IN THE AC.

ARGUMENTS:

N	Integer dimension of the square matrix A.
A	First element of the matrix. For further information, see the write-up for the subroutine GJRDT.
XCH	Vector for record of interchanges used in the elimination. If XCH(I) is non-zero then when row I was used as a pivot row, an interchange with row XCH(I) was necessary. XCH(I) will be zero if no interchange was necessary. XCH will be integral in all cases. If the L*R product is formed the interchanges must be performed in the reverse order to obtain the original matrix, (i.e., interchange rows XCH(N-1) and N-1, ... XCH(1) and 1.)
X	Floating-point switch. 1. Successful decomposition. 0. Overflow, cannot continue.

SUBROUTINES REQUIRED: FSPILL, RSPILL

SYMBOL MANIPULATION - UNPACKING

ENTRY POINTS: DCOMPZ, DZCOMP

PURPOSE: DCOMPZ unpacks a sequence of full BCD words into left-justified BCD characters with five trailing blanks. DZCOMP is the same except that it replaces leading zeroes in a word with blanks.

CALLING SEQUENCES:

```
MAD     EXECUTE DCOMPZ.  (N,A,B)
        EXECUTE DZCOMP.  (N,A,B)
UMAP    CALL DCOMPZ      CALL DZCOMP
        PAR  N           PAR  N
        PAR  A           PAR  A
        PAR  B           PAR  B
```

ARGUMENTS:

N	Integer number of words in A which are to be decomposed.
A	Vector of words to be decomposed. Elements of A are assumed to be stored backwards.
B	Vector into which decomposed characters are placed, one character per word with trailing blanks. Dimension of B must be at least 6N, and B will be stored backwards.

DISMOUNT TAPE

ENTRY POINT: DISMNT

PURPOSE: Prints instructions to the operator to remove a user's tapes, rewinds and unloads the tapes, and stops for removal. (This should be used only for selective removal.)

CALLING SEQUENCES:

```
MAD      EXECUTE DISMNT.(UNIT1,PRO1,UNIT2,PRO2,...,UNITN,PRON)
FORTRAN  CALL DISMNT(UNIT1,PRO1,UNIT2,PRO2,...,UNITN,PRON)
UMAP     CALL DISMNT
         PAR  UNIT1
         PAR  PRO1
         PAR  UNIT2
         PAR  PRO2
         ...  ...
         PAR  UNITN
         PAR  PRON
```

ARGUMENTS: There are two arguments for each tape to be removed.

UNIT	Logical number (full word or fortran integer) of the tape drive from which the tape is to be removed.
PRO	If zero, tape should be stored with file protect ring inserted. If non-zero, tape should be stored with file protect ring removed.

ERROR CONDITION: An attempt to remove a tape which was not requested by the subroutines MOUNT or REPLACE will be treated as an I/O error condition.

SUBROUTINES REQUIRED: .ERR

DOUBLE-PRECISION OPERATIONS

ENTRY POINTS: DFAD, DFSB, DFMP, DFDP, SFDP, DCEXIT

PURPOSE: Perform the double-precision floating-point operations of addition, subtraction, multiplication and division. If the high order word is stored in location Y the low order word must be in location Y+1. The subroutines 'DFAD', 'DFSB', 'DFMP', and 'DFDP' are designed to perform the basic arithmetic operations on double-precision numbers stored in this fashion. The high order word of the result is always normalized and the low order word always has the same sign and a characteristic 27 (base 10) less than the high order word. The entry 'SFDP' ignores the low order word of the divisor by assuming it zero. If the high order word of the divisor is zero the subroutine 'ERROR' is called. This may be avoided by calling 'DCEXIT' beforehand and specifying the instruction to which control is to be transferred.

CALLING SEQUENCES:

```
MAD       EXECUTE *F**.  (A,B,C)
          EXECUTE DCEXIT.  (LOC)
FORTRAN   CALL *F**   (A,B,C)
          CALL DCEXIT (LOC)
UMAP      CALL    *F**,A,B,C
          CALL    DCEXIT,LOC
```

ARGUMENTS:

A	High order word of the addend, subtrahend, multiplicand, or dividend.
B	High order word of the augend, minuend, multiplier, or divisor.
C	High order word of the sum, difference, product, or quotient.
LOC	Instruction to be given control if a division by zero (high order word) is attempted. Fortran users should see point 7 under arguments in the introduction to MAMOS.

NOTE: The current I/O conversion routine .IOH uses these subroutines and hence they will be in core whenever it is.

SUBROUTINES REQUIRED: ERROR.

DOUBLE PRECISION FLOATING POINT ARITHMETIC

ENTRY POINTS: DPFA, DPFM, DPFDV

PURPOSE: Perform double precision arithmetic operations on double precision numbers which are floating point.

CALLING SEQUENCES:

MAD	EXECUTE DPFA.(A1,A2,B1,B2,C1,C2)	Addition
	EXECUTE DPFM.(A1,A2,B1,B2,C1,C2)	Multiplication
	EXECUTE DPFDV.(A1,A2,B1,B2,C1,C2,LOC)	Division
FORTRAN	CALL DPFA (A1,A2,B1,B2,C1,C2)	Addition
	CALL DPFM (A1,A2,B1,B2,C1,C2)	Multiplication
	CALL DPFDV (A1,A2,B1,B2,C1,C2)	Division

UMAP

Addition		Multiplication		Division	
CALL	DPFA	CALL	DPFM	CALL	DPFDV
PAR	A1,T1	PAR	A1,T1	PAR	A1,T1
PAR	A2,T2	PAR	A2,T2	PAR	A2,T2
PAR	B1,T3	PAR	B1,T3	PAR	B1,T3
PAR	B2,T4	PAR	B2,T4	PAR	B2,T4
PAR	C1,T5	PAR	C1,T5	PAR	C1,T5
PAR	C2,T6	PAR	C2,T6	PAR	C2,T6
NORMAL RETURN		NORMAL RETURN		PAR	LOC
				NORMAL RETURN	

ARGUMENTS:

A1	High order part of the first addend, of the first factor, or of the dividend for addition, multiplication or division, respectively.
A2	Low order part paired with A1.
B1	High order part of the second addend, the second factor, or the divisor for addition, multiplication or division, respectively.
B2	Low order part paired with B1.
C1	High order part of the answer resulting from the double precision operation performed on the arguments above.
C2	Low order part of the answer.
LOC	Location for return if zero denominator (in DPFDV only) is detected. (This argument may be omitted. If not given and a zero denominator is detected, then an error comment is printed and control is returned to the system.)
T1,T2, ...,T6	Optional tags.

SUBROUTINES REQUIRED: .EXIT (DPFDV ONLY)

MATRIX MULTIPLICATION USING DOUBLE PRECISION

ENTRY POINTS: DPMAT

PURPOSE: Perform the matrix multiplication A = A * B where A and B are real N*N matrices. Each element of the product matrix is accumulated in double precision and then chopped to single precision. Note that the result is automatically placed in A. B is unchanged by DPMAT.

CALLING SEQUENCES:

FORTRAN	X = DPMAT (N,A,B)
MAD	X = DPMAT.(N,A,B)
UMAP	CALL DPMAT,N,A,B
	RETURN - FLOATING-POINT SWITCH IN AC.

ARGUMENTS:

N	Common integer dimension of the matrices A and B. N may not exceed 99.
A	First element of the matrix A. For further information see the write-up for subroutine GJRDT.
B	First element of the matrix B. For further information see the write-up for subroutine GJRDT.
X	Floating-point switch; 1. Successful multiplication. 0. Overflow, cannot continue.

SUBROUTINES REQUIRED: FSPILL, RSPILL

DOUBLE-PRECISION SQUARE ROOT

ENTRY POINTS: DSQRT

PURPOSE: Form the double-precision floating-point square root of the argument and return it with the sign of the argument, i.e., the square root of a positive number is positive and of a negative number is negative.

CALLING SEQUENCES:

MAD	EXECUTE DSQRT. (A,B)
FORTRAN	CALL DSQRT (A,B)
UMAP	SEE THE SUBROUTINE NASQ.

ARGUMENTS:

A	High order word of the argument, the low order word must be in location A+1.
B	High order word of the signed square root, the low order word will be returned in location B+1.

SUBROUTINES REQUIRED: NASQ

LINEAR EQUATIONS

ENTRY POINTS: DSLE1, DSLE2

PURPOSE: Solve in double-precision the system of simultaneous linear equations with coefficient matrix A of dimension M and right hand side B. The method is Gaussian Elimination with partial pivoting, followed by a single back-substitution. A determinant, which may be scaled as described below, is also computed. Two entries are provided. 'DSLE1' assumes a single-precision input system and converts to double by scaling each equation by its maximum element -- the solution is returned in single-precision. The entry 'DSLE2' assumes a double-precision system, performs no scaling and returns a double-precision solution.

CALLING SEQUENCES:

MAD
```
X = DSLE1. (M,A,B,D)
X = DSLE2. (M,A,B,D)
```

UMAP
```
CALL  DSLE1          CALL  DSLE2
PAR   M              PAR   M
PAR   A              PAR   A
PAR   B              PAR   B
PAR   D              PAR   D
        RETURN WITH X IN THE ACCUMULATOR
```

ARGUMENTS:

M — Integer (MAD type, not FORTRAN) dimension of the coefficient matrix.

A — For 'DSLE2' this argument is ignored. For'DSLE1' it is the first element of the coefficient matrix. It is assumed that the (I,J)-element has linear subscript, relative to A, of (I-1)*M+(J-1). This is standard MAD type storage. The matrix A is not destroyed by either entry. The A-region and the D-region may overlap in storage provided A is not contained in the region D(1)...D(2*M*M).

B — Right hand side and solution vector for both entries. For 'DSLE1' this vector is single-precision but of length at least 2*M. for 'DSLE2' the vector is double-precision IBM 7094 type, i.e., if the high order word of an element is in B(NU) (assuming MAD storage), then the low order word is in B(NU-1). The entry 'DSLE2' assumes that the right hand side of the I-th equation is stored in B(2*I) and B(2*I-1).

D — Both entries return the same information in this region of length at least 2*M*M. The D-region will contain the double-precision L*R factorization of the coefficient matrix. The high order word of the (I,J)-element has a linear subscript relative to D of 2*((I-1)*M+J) and the low order word has this subscript minus 1, (i.e., the elements are IBM 7094 type double-precision). This result should be approximately the same as that produced by "GAUSS". Note restriction (5).

LINEAR EQUATIONS (CONTINUED)

'DSLE1' ignores the contents of the D-region and immediately sets up the scaled coefficient matrix with subscripts as described above. 'DSLE2' assumes that the double-precision coefficient matrix is stored in the D-region in the manner described above.
For both entries the number D is taken as a scale factor for the determinant and the high order word of the determinant is returned in D. If the input value of D is zero then the determinant will be returned as zero. This will not cause, however, a singularity return. A double-precision IBM 7094 type determinant is available in erasable, the high order word in 77776 and the low in 77777. Note restriction (4).

X Returns
(1) 1. - Successful computation.
(2) 0. - Overflow, cannot continue.
(3) -1. - Singular matrix.
(4) -2. - Zero row in matrix (impossible with DSLE2).

RESTRICTIONS:

(1) 'RSPILL' is called immediately before the return.
(2) 'DCEXIT' is called just before the return and set so that 'ERROR' will be called if division by zero is attempted.
(3) The matrix A and the vector D may occupy the same storage region, provided the region reserved for A is not contained in that reserved for D.
(4) The sign of the low order determinant word in 77777 may have the wrong sign. The high order word in 77776 has the correct sign.
(5) In performing the L*R factorization which is stored in the D-region, physical interchanges are not used, only row address vector entries, i.e.,
R(1)...R(M), where R(I)=(I-1)*M.
Consequently the rows of the matrix are jumbled. The row address vector is stored in erasable and has base address 77764.

SUBROUTINES REQUIRED: DCEXIT, DFDP, DFMP, DFSUB, ERROR, FSPILL, RSPILL, SFDP.

EIGENVALUES AND EIGENVECTORS

ENTRY POINTS: EIGN

PURPOSE: To compute all eigenvalues and/or eigenvectors of a real symmetric matrix by the Jacobi method. The elements of the matrix must satisfy the condition SM = SUM((A(I,J)/AM).P.2) .LE.2.P.257, where AM = MIN/A(I,J)/ .NE.0.

CALLING SEQUENCES:

```
MAD       S=EIGN.(A,N,K,V,F)
FORTRAN   S=EIGN(A,N,K,V,F)
UMAP      CALL    EIGN
          PAR     A
          PAR     N
          PAR     K
          PAR     V
          PAR     F
          NORMAL RETURN-S IN THE ACCUMULATOR.
```

ARGUMENTS:

A — First element of a floating point array in which the matrix is stored. There must be no extra locations between the rows or columns in this array. In MAD this is accomplished by setting the third entry of the dimension vector for A equal to N before reading the data. In FORTRAN the original dimension statement would have to specify an array dimension equal to N by N. In all cases (MAD, UMAP, FORTRAN) the spaces may be eliminated by considering the matrix as a linear array with the rows or columns stored sequentially. The original A array is destroyed by the computation and the eigenvalues replace the diagonal elements of the A matrix.

N — Integral order of the matrix A.

K — Integer switch: If K=0, no eigenvectors desired. If K=1, eigenvectors are to be computed.

V — First element of a floating point array in which the eigenvectors are to be stored. As with the A array, V is treated as a linear vector by EIGN. The first vector corresponding to the first diagonal element of A will be stored in the first N locations of V. The second vector in the second N, etc. If K=0, V is a dummy argument.

F — If either SM.GE.2.P.128 or AM.P.2.LE.2.P.-123, but the restriction stated in the purpose is not violated, then a return is made with S = 2.0 (see S) and the elements of the original array are scaled with an appropriate scale factor which is stored in F.

S — Computation switch - floating point.
S=1.0 Error return - SM.GE.2.P.257.
S=2.0 Successful computation but the matrix (and the eigenvalues) has been scaled by a factor stored in F.
S=3.0 Normal return, successful computation.

FLOATING POINT LOGARITHM

ENTRY POINTS: ELOG

PURPOSE: Compute LN(X) for floating point argument X.G.0.

CALLING SEQUENCES:

```
MAD       Y = ELOG.(X,LOC)
FORTRAN   Y = ELOG(X)
UMAP      CALL    ELOG
          PAR     X,T
          PAR     LOC
          NORMAL RETURN - Y IN THE ACCUMULATOR.
```

ARGUMENTS:

X	Floating point argument for which the log (to the base E) is desired.
Y	The floating point log of the argument X.
LOC	Location for return if error detected. (This argument may be omitted.)
T	Optional tag.

ERROR CONDITIONS:

If X .LE. 0. the error routine is initiated. If LOC is given, control is returned to the caller. Otherwise,

'****ELOG ARG NEGATIVE OR ZERO'

is printed on the output tape and control is returned to the system, giving a dump of core if requested.

SUBROUTINES REQUIRED: .EXIT

EXIT SUBROUTINE

ENTRY POINTS: .EXIT

PURPOSE: Provides error return to system from non-I/O subroutines.

CALLING SEQUENCE:

```
    UMAP     CLA  N+1,4
             CALL .EXIT
             BLK  COMENT,T,L
             PAR  XR4
                  ...
      COMENT BCI  L,C...
```

ARGUMENTS:

N	Number of arguments for the subroutine not including the optional error return argument.
COMENT	Location of first word of comment to be printed.
T	If zero, an optional error return is not permitted, and 'CLA N+1,4' is not needed. If non-zero, the subroutine user may specify an optional error return.
L	The number of six letter machine words making up the comment.
XR4	Optional argument containing 2's complement of number XXXXX to be printed out with additional comment 'AT LOCATION XXXXX'.
C	Carriage control for comment line, which is followed by the comment.

NOTE: Index register 4 is assumed to have been restored to its condition at the time of doing the "CLA N+1,4' (not necessary if no error return).

SUBROUTINES REQUIRED: SPRINT, ATLOC, ERROR

EXPONENTIATION - THE BASE E

ENTRY POINTS: EXP

PURPOSE: Raise E to the floating point exponent X.

CALLING SEQUENCES:

```
MAD       Y = EXP.(X,LOC)
FORTRAN   Y = EXP (X)
UMAP      CALL    EXP
          PAR     X,T
          PAR     LOC
          NORMAL RETURN - Y IN THE ACCUMULATOR.
```

ARGUMENTS:

X	The floating point power to which E is to be raised.
Y	Result in floating point.
LOC	Location for return if error detected. (This argument may be omitted.)
T	Optional tag.

ERROR CONDITIONS:

If X .G. 87.3, the result will exceed machine size, and the error procedure is begun. If LOC is given, then control is returned to that location. If LOC is not given, the comment '***EXP ARG EXCEEDS 87.3' is printed, followed by a dump (if requested).

SUBROUTINES REQUIRED: .EXIT

ERROR FUNCTION SUBROUTINE

ENTRY POINTS: ERF

PURPOSE: To compute ERF(X), where ERF(X) is defined as ERF(X) = 2/SQRT.(PI) times the integral from zero to X of EXP (-(T.P.2)) DT where ERF(-X) = -ERF(X).
For a shorter, faster, less accurate subroutine, see ERRFN.

CALLING SEQUENCES:

MAD	Y = ERF.(X)
FORTRAN	Y = ERF(X)
UMAP	CALL ERF,X
	RETURNS WITH VALUE IN ACCUMULATOR

Both X and Y are floating point.

ACCURACY:
Plus or minus 2 in eighth significant digit.

Adapted from Ames Research Center AL-ERF (Share Dist. no. 836)

SUBROUTINES USED: EXP

ERROR FUNCTION SUBROUTINE

ENTRY POINTS: ERRFN, FREQ

PURPOSE: To compute the error function, ERF, (for definition see the write-up on ERF) and the normal frequency function. For a more accurate, but longer and slower routine, see ERF.

CALLING SEQUENCES:
Error Function -

MAD	Y = ERRFN.(X)
FORTRAN	Y = ERRFN(X)
UMAP	CALL ERRFN,X
	RETURNS VALUE IN ACCUMULATOR

Normal Frequency Function -

MAD	Z = FREQ.(X)
FORTRAN	Z = FREQ(X)
UMAP	CALL FREQ,X
	RETURNS VALUE IN ACCUMULATOR

X,Y, and Z are floating point.

ACCURACY:
Plus or minus 1 in fifth significant digit.

Adapted from Share Dist. no. 897

EXPONENTIATION - INTEGER BASE AND INTEGER EXPONENT

ENTRY POINTS: EXP1

PURPOSE: Raise an integer base to an integer power.

CALLING SEQUENCES:

```
    UMAP      CALL    EXP1
              PAR     I,T1
              PAR     J,T2
              PAR     LOC
              NORMAL RETURN - N IN THE ACCUMULATOR.
```

ARGUMENTS:

I	Integer base.
J	Integer exponent.
T1,T2	Optional tags.
LOC	Location for return if error found (this argument may be omitted.)
N	Result. If I = 0 or J.L.0 or (I = 0 and J = 0), N = 0. If J = 0 and I.NE.0, N = 1.

ERROR CONDITION:

If N.GE.2.P.35, the error procedure is initiated. If LOC is given, control is returned to the caller. Otherwise, the comment, 'EXP1 ANS TOO LARGE' will be printed and a dump given if requested by the programmer.

SUBROUTINES REQUIRED: .01311, .EXIT

EXPONENTIATION - FLOATING POINT BASE AND INTEGER EXPONENT

ENTRY POINTS: EXP2

PURPOSE: Raise a floating point number to an integer power.

CALLING SEQUENCES:

```
    UMAP      CALL    EXP2
              PAR     B,T1
              PAR     J,T2
              PAR     LOC
              NORMAL RETURN - X IN THE ACCUMULATOR.
```

ARGUMENTS:

B	Floating point base.
J	Integer exponent.
T1,T2	Optional tag.
LOC	Location for return if error detected. (This argument may be omitted.)
X	Result. If B = 0, X = 0. If J = 0 and B.NE.0, X = 1.

ERROR CONDITION:

If X is out of range of machine size, the error procedure is initiated. If LOC is given, control is returned to the caller. Otherwise, the comment 'EXP2 ANS OUT OF RANGE' is printed and a dump is given if requested by the programmer.

SUBROUTINES REQUIRED: .01301, .EXIT

EXPONENTIATION - FLOATING POINT BASE AND FLOATING POINT EXPONENT

ENTRY POINT: EXP3

PURPOSE: Raise a floating point number to a floating point power.

CALLING SEQUENCES:

```
    UMAP     CALL     EXP3
             PAR      B,T1
             PAR      A,T2
             NORMAL RETURN - X IN THE ACCUMULATOR.
```

ARGUMENTS:

B	Floating point base.
A	Floating point exponent.
T1,T2	Optional tags.
X	B.P.A in floating point. If B = 0, X = 0.

ERROR CONDITIONS:
If the base is negative and the exponent is non-integral, the comment, 'EXPONENTIATION ERROR - NEGATIVE BASE, NON-INTEGRAL EXPONENT, OR ANS TOO LARGE' is printed and a dump is given if requested.

SUBROUTINES REQUIRED: .EXIT, .01300, SQRT, ELOG, EXP

FLOATING POINT SPILL ROUTINE

ENTRY POINTS: FSPILL, RSPILL

PURPOSE: The entry 'FSPILL' turns on the floating-point trap mode indicator and provides for the following action in the event of a floating-point spill.

Underflow -- Set either or both of the registers causing underflow to zero and continue the computation from the point at which the floating-point trap occurred.

Overflow -- Transfer to the instruction specified by the argument in the last call of 'FSPILL'. The transfer is accomplished without destroying any of the internal registers or indicators.

This entry also saves the contents of location 8 (the floating-point trap location) if 'FSPILL' has not been called before or if 'RSPILL' has been called since the last call of 'FSPILL'. The entry 'RSPILL' restores the contents of location 8 that were previously saved by 'FSPILL' and turns on a switch so that 'FSPILL' will save location 8 the next time it is called. Until the next call of 'FSPILL' the 'RSPILL' entry will act as a dummy subroutine. both entries preserve all indicators and all internal registers except the MQ register (this latter is also preserved unless something must happen to location 8).

CALLING SEQUENCES:

```
    MAD    EXECUTE FSPILL. (LOC)
           EXECUTE RSPILL.
```

FLOATING-POINT SPILL ROUTINE (CONTINUED)

```
    FORTRAN  CALL FSPILL (LOC)
             CALL RSPILL
    UMAP     CALL     FSPILL,LOC
             CALL     RSPILL
```

ARGUMENTS:

LOC — Instruction to be given control if an overflow occurs. Fortran users should see point 7 under arguments in the introduction to this section.

FLOATING-POINT UNDERFLOW SWITCH

ENTRY POINTS: FTRAP, NTRAP

PURPOSE: Normally, a floating point underflow is considered an error and control is returned to the system. If FTRAP is executed prior to the underflow, the number that caused the underflow will be set to zero and computation will proceed. If NTRAP is executed, underflow will again be treated as an error.

CALLING SEQUENCE:

```
    MAD      EXECUTE FTRAP.
             EXECUTE NTRAP.
    UMAP     CALL   FTRAP
             CALL   NTRAP
    FORTRAN  CALL FTRAP
             CALL NTRAP
```

FLOATING POINT GAMMA FUNCTION

ENTRY POINTS: GAMMA

PURPOSE: Compute GAMMA(X) for floating point argument X.

CALLING SEQUENCES:

```
    MAD      Y = GAMMA.(X)
    FORTRAN  Y = GAMMA(X)
    UMAP     CALL GAMMA,X
             STO  Y
```

ARGUMENTS:

X — Floating point argument.

Y — Floating point result.

FLOATING POINT GAMMA FUNCTION (CONTINUED)

COMMENTS:

1. If X is zero or a negative integer, then GAMMA(X) = .1E35.
2. "SAFE" range of argument for GAMMA function is -34.2 .LE. X .LE. 34.4. Outside these values, floating point overflow or underflow will occur.
3. For Algorithm, see Algorithm No. 31, COMM. ACM, FEBR, 1961.

MATRIX FACTORIZATION BY L-R DECOMPOSITION

ENTRY POINT: GAUSS

PURPOSE: Factorization of an arbitrary real matrix -A- into a product of an upper triangular matrix -R- and a monic lower triangular matrix -L- by Gaussian Elimination with interchanges, i.e., A = L * R. The matrix -L- has the further property that the absolute value of L(I,J) is less than or equal to 1. This decomposition is unique. The matrix -R- is stored in the upper half of the original matrix and -L- in the lower half, the diagonal belonging to the matrix -R-. See subroutine CROUTP.

CALLING SEQUENCES:

```
FORTRAN  X = GAUSS  (N,A,XCH)
MAD      X = GAUSS. (N,A,XCH)
UMAP     CALL   GAUSS,N,A,XCH
         RETURN - X WILL BE IN THE AC.
```

ARGUMENTS:

N — Integer dimension of the square matrix A.

A — First element of the matrix. For further information see the write-up for the subroutine GJRDT.

XCH — Vector for record of interchanges used in the elimination. If XCH(I) is non-zero then when row I was used as a pivot row, an interchange with row XCH(I) was necessary. XCH(I) will be zero if no interchange was necessary. XCH will be integral in all cases. If the L*R product is formed the interchanges must be performed in the reverse order to obtain the original matrix, i.e., interchange rows

XCH(N-1) AND N-1,
.....
XCH(1) AND 1.

X — Floating-point switch.

1. Successful decomposition.
0. Overflow, cannot continue.

SUBROUTINES REQUIRED: FSPILL, RSPILL

SIMULTANEOUS LINEAR EQUATIONS BY MATRIX INVERSION

ENTRY POINTS: GJRDT

PURPOSE: Computes (M-N) solution vectors of a set of N simultaneous real linear equations in N unknowns. The inverse of the coefficient matrix is automatically produced, as is the determinant. The method used is a Gauss-Jordan reduction of an arbitrary augmented matrix upon itself using a complete pivotal strategy.

CALLING SEQUENCE:

FORTRAN	X = GJRDT (N,M,A,D)
MAD	X = GJRDT. (N,M,A,D)
UMAP	CALL GJRDT,N,M,A,D
	RETURN - X WILL BE IN THE AC.

ARGUMENTS:

N — Number of equations, i.e., the row dimension of the matrix. N must be an integer and less than 100.

M — M = N + (the number of solution vectors desired), i.e., the column dimension of the matrix. Notice that if N = M, no solution vectors are computed, but the matrix is inverted. M must be an integer.

A — The highest location in core that contains an element of the matrix - the location of the A(1,1) element where

$$A = \begin{pmatrix} A(1,1) & \dots & A(1,N) & \dots & A(1,M) \\ & & \cdots\cdots & & \\ A(N,1) & \dots & A(N,N) & \dots & A(N,M) \end{pmatrix}$$

The matrix must be stored backwards by rows and must be packed. A(1,1) must be the highest location and proceeding in the direction of decreasing storage, addresses must be A(1,2)...A(1,M),A(2,1)...A(2,M),...,A(N,M) without any gaps. This mode of storage is normal MAD storage.

D — On return from GJRDT the determinant of A will be in this location.

X — Floating-point switch.

- 1. Successful computation.
- -1. Singular matrix. The condition for singularity is that the determinant be zero.

SIMULTANEOUS LINEAR EQUATIONS BY MATRIX INVERSION

ENTRY POINTS: GJRDTP

PURPOSE: Inverts the matrix and computes (M-N) solution vectors of a set of N simultaneous real linear equations in N unknowns. The determinant of the matrix is automatically computed. GJRDTP performs a Gauss-Jordan reduction of an arbitrary augmented matrix upon itself using a partial pivotal strategy. For most systems GJRDTP and GJRDT will produce results which differ only in the digits subject to round-off error. Unfortunately there is no clear cut rule as to which subroutine to use for a given set of equations. In any case, GJRDT is the most reliable.

CALLING SEQUENCE:

```
FORTRAN  X = GJRDTP  (N,M,A,D )
MAD      X = GJRDTP. ( N,M,A,D )
UMAP     CALL    GJRDTP,N,M,A,D
         RETURN - X WILL BE IN THE AC.
```

ARGUMENTS:

For a description of the arguments, see the write-up for subroutine GJRDT.

HARMONIC ANALYSIS

ENTRY POINTS: HAS1, HAS1S

PURPOSE: Given a set of points Y(I) (I = 0,1,2,...,K-1) corresponding to a set of equally spaced arguments X(I), this subroutine computes the coefficients A(I), B(I), C(I), D(I) of the following series

$$Y(X) = A(0) + SUM(A(N)*COS(NX) + B(N)*SIN(NX))$$
$$Y(X) = A(0) + SUM(C(N)*SIN(NX + D(N)))$$

M is the parameter designating the number of harmonics and M.LE.K/2 if K is even or M.LE.(K-1)/2 if K is odd. The function Y(X) is assumed to be periodic of period 2PI with Y(0) = Y(K).

CALLING SEQUENCES:

MAD
```
EXECUTE HAS1.(K,M,Y,A,S)
EXECUTE HAS1S.(K,M,Y,A)
```

FORTRAN
```
CALL HAS1(K,M,Y,A,S)
CALL HAS1S(K,M,Y,A)
```

UMAP

CALL	HAS1	CALL	HAS1S
PAR	K,T1	PAR	K,T1
PAR	M,T2	PAR	M,T2
PAR	Y	PAR	Y
PAR	A	PAR	A
PAR	S	NORMAL	RETURN
NORMAL	RETURN		

ARGUMENTS:

K	Integral number of points Y(I).
M	Integral number of harmonics desired.
Y	First element of a floating point vector in which the input points are stored.
A	First element of a floating point vector in which HAS1 will store the answers. This array must be of length at least 7 + 5M. The answers are stored in groups of 5 beginning at A as .....,A(N),B(N),C(N),D(N),C(N)/C(MAX).
S	Temporary storage region supplied by the programmer of length at least 2K. This region is used by HAS1 to store values for Sine and Cosine. HAS1S computes values for Sine and Cosine every time that they are needed. Both subroutines will produce the same results.
T1,T2	Optional tags.

SUBROUTINES REQUIRED: SIN, COS, ATN1, SQRT

MATRIX INVERSION

ENTRY POINT: IBDS

PURPOSE: This subroutine is a modification of BORDS that assumes a FORTRAN calling program. An extra argument has been added as a result of the FORTRAN method of dimensioning. For further information, see the write-up for subroutine BORDS.

CALLING SEQUENCE:

```
FORTRAN  IX = IBDS ( N,M,A,D )
```

ARGUMENTS:

N	FORTRAN integer giving the true dimension of the matrix. N must be less than 150.
M	FORTRAN integer giving the FORTRAN dimension of the matrix.
A	Name of the matrix. The matrix -A- is assumed to conform to normal FORTRAN two dimensional array storage.
D	On return from IBDS, D will contain the determinant of -A-.
IX	FORTRAN integer switch. 1 Successful inversion. -K Negative integer giving the dimension of the smallest principal minor equal to zero.

INCOMPLETE ELLIPTIC INTEGRALS

ENTRY POINTS: IEF1

PURPOSE: Given the amplitude A and the modulus B, to evaluate the incomplete elliptic integrals of the first and second kind.

CALLING SEQUENCES:

```
MAD       EXECUTE IEF1.(A,B,E,F,G)
FORTRAN   CALL IEF1(A,B,E,F,G)
UMAP      CALL    IEF1
          PAR     A
          PAR     B
          PAR     E
          PAR     F
          PAR     G
          NORMAL RETURN
```

ARGUMENTS:

A	Floating point amplitude A of the integral.
B	Floating point modulus B of the integral.
E	The incomplete elliptic integral of the second kind, E(A,B), in floating point.
F	The incomplete elliptic integral of the first kind, F(A,B), in floating point.
G	Computation switch. (floating point) G = 1.0 Normal return, computation successful. G = 2.0 B out of range , B.G.8.99985. G = 3.0 A out of range , A.L.0 or A.G.PI/2

SUBROUTINES REQUIRED: SIN, SQRT

SET IOH FIELD SIZE ERROR CONDITION

ENTRY POINTS: IOHSIZ

PURPOSE: Normally, if in a format the user gives a field size too small for an integer or floating point number, this is considered an error and execution is terminated. Use of IOHSIZ allows this number to be printed (or punched) in a truncated form, and for execution then to continue.

CALLING SEQUENCES:

```
MAD       EXECUTE IOHSIZ.(N)
FORTRAN   CALL IOHSIZ (N)
UMAP      CALL  IOHSIZ
          PAR   N
```

ARGUMENTS:

N — If N contains zero, then thereafter, when a number too large for the field width occurs, normal procedure will be followed, i.e., a comment will be printed and the job will be terminated. If the contents of N are non-zero, then thereafter an oversized number for an E,F, or I field will be punched or printed. It will be right justified with the left end, including the sign, truncated. Execution will then continue.

SUBROUTINES REQUIRED: .ERR, SPRINT, SKIP6, .03311, DFMP, DFDP

NUMERICAL INTEGRATION OF SINGLE OR MULTIPLE INTEGRALS

ENTRY POINTS: ITINT

PURPOSE: To compute:

$$\int_{A_0}^{B_0} \int_{A_1(X_0)}^{B_1(X_0)} \cdots \int_{A_{k-1}(X_0,\ldots,X_{k-2})}^{B_{k-1}(X_0,\ldots,X_{k-2})} F_{k-1}(X_0,\ldots,X_{k-2},X_{k-1})\,DX_{k-1}\cdots DX_0$$

for any integer K.GE.1 by Gaussian Quadrature method. A value for the L-th integral in the above expression is computed from the sum;

$$\int_{A_L}^{A_L+H} F_L\,DX_L + \int_{A_L}^{A_L+H} F_L+DX_L + \cdots + \int_{A_L+(N_L-1)*H}^{A_L+N_L*H} F_L\,DX_L$$

where N_L may be a function of $X_0,\ldots,X_{L-1}$ and $H = (B_L-A_L)/N_L$ and where

$$\int_{A_L+T*H}^{A_L+(T+1)*H} F_L\,DX_L$$

is approximated from M_L evaluations (possibly a function of $X_0,\ldots,X_{L-1}$) of F_L on the interval $A_L+T*H.L.X_L.L.A_L+(T+1)*H$ (all other X's held fixed. M greater than 1, less than 9.) This makes necessary many entrances into ITINT and many exits from it; one exit and entrance for each evaluation of each F_L, one initial entrance and one final exit.

In order to tell the user what is to be done on each exit, ITINT takes on an integer value in the range 0,1,2,...,K-1,K depending on whether a value for one of the F's is to be computed or the integration is complete.

USE: In MAMOS the use of ITINT would conform to the following rough drafts:

MAD USAGE.

```
INTEGER K, ITINT.
DIMENSION Q('10K'), F('K-1'), A('K-1'), B('K-1'), ZN('K-1'),
1   ZM('K-1'), X('K-1')
- - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - (Set up values at least for
- - - - - - - - - - - - - - -  ZN(0), ZM(0),A(0),B(0)
```

NUMERICAL INTEGRATION OF SINGLE OR MULTIPLE INTEGRALS (CONTINUED)

(MAD USAGE - CONTINUED)

```
    INTEG TRANSFER TO S(ITINT.(F,A,B,ZN,ZM,Q,K,X))
    S(0)  - - - - - - - - - - - - - - - -
          - - - - - - - - - - - - - - - - (Compute a value for F(0), and possibly
          - - - - - - - - - - - - - - - -  set up values for ZN(1),ZM(1),A(1),B(1))
          TRANSFER TO INTEG
    S(2)  - - - - - - - - - - - - - - - -
          - - - - - - - - - - - - - - - - (Compute a value for F(1), and possibly
          - - - - - - - - - - - - - - - -  set up values for ZN(2),ZM(2),A(2),B(2))
          TRANSFER TO INTEG


    S(/K-1/)- - - - - - - - - - - - - - -
          - - - - - - - - - - - - - - - - (Compute a value for F(K-1))
          - - - - - - - - - - - - - - - -
          TRANSFER TO INTEG
    S(/K/)- - - - - - - - - - - - - - - -
          - - - - - - - - - - - - - - - - (Integration complete, answer in Q(0))
          - - - - - - - - - - - - - - - -
          'K-1' stands for an integer constant whose value is at least as
          great as K's value reduced by one (and similarly for '10K'), and
          /K-1/ stands for the integer constant having K's value minus one
          (and similarly for /K/).

    EXAMPLE:

          INTEGER ITINT.
          DIMENSION Q(20),F(1),A(1),B(1),ZN(1),ZM(1),X(1)
    START READ DATA  (ZN(0),ZN(1),ZM(0),ZM(1),A(0),B(0))
    INTEG TRANSFER TO S(ITINT.(F,A,B,ZN,ZM,Q,2,X))
    S(0)  F(0) = X(0)
          A(1) = -X(0)
          B(1) = X(0)
          TRANSFER TO INTEG
    S(1)  F(1) = SIN.(X(0)*X(1))
          TRANSFER TO INTEG
    S(2)  PRINT RESULTS Q(0)
          TRANSFER TO START
          END OF PROGRAM
```

FORTRAN USAGE.

```
          DIMENSION Q('10K+1'),A('K'),B('K'),F('K'),ZN('K'),ZM('L'),X('K')
          - - - - - - - - - - - - - - - -
          - - - - - - - - - - - - - - - - (Set up values for A(1),B(1),ZN(1),
          - - - - - - - - - - - - - - - -  ZM(1) at least)
    1     I = ITINT(F,A,B,ZN,ZM,Q,K,X)) + 2
          GO TO (2,3,...,/K+1/,/K+2/), I
    2     - - - - - - - - - - - - - - - -
          - - - - - - - - - - - - - - - - (Compute F(1), and possibly set up
          - - - - - - - - - - - - - - - -  values for A(2), B(2), ZN(2), ZM(2))
          GO TO 1
```

NUMERICAL INTEGRATION OF SINGLE OR MULTIPLE INTEGRALS (CONTINUED)

(FORTRAN USAGE - CONTINUED)

```
3       - - - - - - - - - - - - - - - - (Compute a value for F(2), and possibly
        - - - - - - - - - - - - - - - -  set up values for A(3), B(3), ZN(3), ZM(3))
        - - - - - - - - - - - - - - - -
        GO TO 1


/K+1/   - - - - - - - - - - - - - - - -
        - - - - - - - - - - - - - - - - (Compute a value for F(K))
        - - - - - - - - - - - - - - - -
        GO TO 1
/K+2/   - - - - - - - - - - - - - - - -
        - - - - - - - - - - - - - - - - (Integration complete, answer is Q(1))
        - - - - - - - - - - - - - - - -
```

(Since FORTRAN does not allow zero subscripts, FORTRAN users should, while reading the method section, mentally increase all subscripts by one. The notation '10K=1' etc. is explained below the rough draft for MAD users.)

EXAMPLE:

```
        DIMENSION Q(21),F(2),A(2),B(2),ZN(2),ZM(2),X(2)
777     READ INPUT TAPE 7, 5, A(1),B(1),ZN(1),ZN(2),ZM(1),ZM(2)
5       FORMAT (6F10.3)
1       I = ITINT(F,A,B,ZN,ZM,Q,2,X) + 2
        GO TO (2,3,4), I
2       F(1) = X(1)
        A(2) = -X(1)
        B(2) = X(1)
        GO TO 1
3       F(2) = SIN(X(1)*X(2))
        GO TO 1
4       WRITE OUTPUT TAPE 6, 6, Q(1)
6       FORMAT (E15.6)
        GO TO 777
        END
```

UMAP USAGE.

```
        - - - - - - - - - - - - - - - - (Set up values at least for
        - - - - - - - - - - - - - - - -  A, B, ZN, ZM)
        - - - - - - - - - - - - - - - -
INTEG   CALL    ITINT
        PAR     F
        PAR     A
        PAR     B
        PAR     ZN
        PAR     ZM
        PAR     Q
        PAR     K
        PAR     X
```

NUMERICAL INTEGRATION OF SINGLE OR MULTIPLE INTEGRALS (CONTINUED)

(UMAP USAGE - CONTINUED)

```
          PAX     0,4
          TRA     LOCVEC,4
          TRA     LK
          TRA     LKM1
           .
           .
           .
          TRA     L2
          TRA     L1
LOCVEC    - - - - - - - - - - - - - -
          - - - - - - - - - - - - - -   (Compute value for F, and possibly set
          - - - - - - - - - - - - - -    up values for ZN-1,ZM-1,A-1,B-1)
          TRA     INTEG
L1        - - - - - - - - - - - - - -   (Compute a value for F-1, and possibly
          - - - - - - - - - - - - - -    set up values for ZN-2,ZM-2,A-2,B-2)
          - - - - - - - - - - - - - -
          TRA     INTEG
           .
           .
           .
LKM1      - - - - - - - - - - - - - -   (Compute a value for F-C(K)=1, where
          - - - - - - - - - - - - - -    C(K) represents the contents of
          - - - - - - - - - - - - - -    location K.)
          TRA     INTEG
LK        - - - - - - - - - - - - - -
          - - - - - - - - - - - - - -   (integration complete, answer in Q)
          - - - - - - - - - - - - - -
           .
           .
           .
Q         BTS     '10K+1'
F         BTS     'K'
A         BTS     'K'
B         BTS     'K'
ZN        BTS     'K'
ZM        BTS     'K'
X         BTS     'K'
```

(For an explanation of 'K' and '10K+1' see the rough draft for MAD.)

ARGUMENTS:

Q The name of the first location of a floating-point "scratch vector". Upon completion of the integration, location Q will contain the value of the integral.

F The name of the first element of a floating-point vector in which the evaluations of the integrands (that is, the F's -- see method) are stored.

A,B ZN,ZM The name of the first elements of the floating-point vectors in which the A's (lower limits of the component integrals), B's (upper limits), N's (number of subintervals into which each component interval of integration (A,B) is to be divided), and

NUMERICAL INTEGRATION OF SINGLE OR MULTIPLE INTEGRALS (CONTINUED)

M's (number of evaluations to be made of each integrand function on each subinterval) -- see method -- are stored. The values in ZN and ZM need not be integral numbers. The integer parts of their values will be used.

K An integer variable or constant giving the multiplicity of the integral to be solved (K .GE. 1).

X The location of the first element of a floating-point vector in which the current values for the X's have been stored by ITINT prior to each exit. All evaluations of each integrand function, F, are to be made using these values.

TAPE LABELING

ENTRY POINTS: LABEL

PURPOSE: Writes a label on a user's tape.

CALLING SEQUENCES:

MAD	EXECUTE LABEL.(NAME,UNIT,LOAD,DENS,MODE)
FORTRAN	CALL LABEL (NAME,UNIT,LOAD,DENS,MODE)
UMAP	CALL LABEL,NAME,UNIT,LOAD,DENS,MODE

ARGUMENTS:

NAME	One word BCD name to be used in label.
UNIT	Logical number (full word or FORTRAN integer) of the tape drive on which the tape to be labeled is mounted.
LOAD	If zero, single load point. If non-zero, double load point.
DENS	If zero, tape is in low density. If non-zero, tape is in high density. (This should normally be non-zero.)
MODE	If zero, BCD. If non-zero, binary.

SUBROUTINES REQUIRED: BNBCD, .ERR

SHIFTING OPERATIONS

ENTRY POINTS: LSH, RSH

PURPOSE: Provide shifting of single words by arbitrary number of binary digits-

CALLING SEQUENCES:

MAD	X = LSH.(A,N)	(Left shift)
MAD	X = RSH.(A,N)	(Right shift)

ARGUMENTS:

A	Word to be shifted.
N	Integer number of binary shifts.
X	Result of shifting A by N bits to the left (LSH.) or to the right (RSH.). Vacated positions are filled with zeroes.

TAPE MOUNTING

ENTRY POINTS: MOUNT

PURPOSE: Prints instructions to the operator to mount a user's tapes, stops for mounting, and then checks the tapes' labels to see if the correct tapes were mounted.

CALLING SEQUENCES:

MAD
```
DATE=MOUNT.(NUM1,NAME1,UNIT1,DENS1,PRO1,MODE1,NUM2,NAME2,
UNIT2,DENS2,PRO2,MODE2,...,NUMN,NAMEN,UNITN,DENSN,PRON,MODEN)
```

FORTRAN
```
DATE=MOUNT(NUM1,NAME1,UNIT1,DENS1,PRO1,MODE1,NUM2,NAME2,
UNIT3,DENS2,PRO2,MODE2,...,NUMN,NAMEN,UNITN,DENSN,PRON,MODEN)
```

UMAP
```
CALL MOUNT
PAR  NUM1
PAR  NAME1
PAR  UNIT1
PAR  DENS1
PAR  PRO1
PAR  MODE1
...  ...
PAR  NUMN
PAR  NAMEN
PAR  UNITN
PAR  DENSN
PAR  PRON
PAR  MODEN
DATE RETURNED IN THE ACCUMULATOR
```

ARGUMENTS: There are six arguments for each tape to be mounted.

NUM — Number (full word or FORTRAN integer) assigned to tape. If zero, the subroutine LABEL is called using NAME, DENS and MODE as arguments. If negative and non-zero, the subroutine LABEL is called and a double load point is requested.

NAME — One word BCD name used in tape label.

UNIT — Logical number (full word or FORTRAN integer) of the tape drive on which the tape is to be mounted.

DENS — If zero, tape is in low density. If non-zero, tape is in high density. (This should normally be non-zero.)

PRO — If zero, file protect ring should be inserted. If non-zero, file protect ring should be removed.

MODE — If zero, BCD. If non-zero, binary.

DATE — The BCD date in the label. It has the form DD/MM where DD is the day of the month MM. If there is more than one group of arguments, the date will be that on the tape specified by the last group.

MOVE ARRAYS

ENTRY POINTS: MOVER

PURPOSE: Moves or reverses linear arrays.

CALLING SEQUENCE:

```
    MAD       EXECUTE MOVER.(L1,L2,.....,LN)
```

ARGUMENTS: The ,LI, are of the form ,A, or ,A...B, . The arguments are used by a MOVER in pairs, as follows:

I. ,A...B,C...D,
The contents of A through B are moved into C through D. /C-D/+1 words are moved. The contents of A through B are unchanged.

II. ,A,B,
The contents of A through B are reversed. (I.e., the contents of A through B are moved into B through A.)

DOUBLE-PRECISION SQUARE ROOT

ENTRY POINTS: NASQ

PURPOSE: Form the double-precision square root of the absolute value of the double-precision floating-point number in the AC and MQ registers, the high order word occupying the AC. Return the square root in the AC and MQ registers, in the same fashion, with the sign of the original number. Three single-precision and two double-precision Newton-Raphson iterations are used, the single-precision operation on an argument scaled to lie in the range .5 to 2.

CALLING SEQUENCE:

```
    UMAP      (Place argument in AC and MQ if it is not already there.)
              CALL    NASQ

              (Result will be in AC and MQ registers.)
```

SUBROUTINES REQUIRED: DFAD, DFDP

NORMALLY DISTRIBUTED RANDOM NUMBER GENERATOR

ENTRY POINTS: NDRN1A, NDRN1B, NDRN1C, NDRN1D

PURPOSE: Produce a random number such that a set of such numbers will have a specified mean and standard deviation.

CALLING SEQUENCES:

```
MAD       EXECUTE NDRN1B.(A,B,C)
FORTRAN   CALL NDRN1B(A,B,C)
UMAP      CALL    NDRN1B
          PAR     A
          PAR     B
          PAR     C
          NORMAL RETURN
```

ARGUMENTS:

A	Floating point standard deviation of the desired distribution.
B	Floating point mean of the desired normal distribution.
C	Floating point random number.

SPECIAL ENTRIES: NDRN1B uses an integer parameter J to compute each random number. J is initially set at (2.P.35-1) and changes with each execution of NDRN1B. The following entries enable the programmer to pick up the current value of J and to use it as input to NDRN1B in order to initialize a sequence of random numbers during execution of the calling program.

1. PURPOSE: Save the current value of J that would have been used to calculate the next random number.

CALLING SEQUENCES:

```
MAD       EXECUTE NDRN1D. (I)
FORTRAN   CALL NDRN1C(K)
UMAP      CALL    NDRN1D
          PAR     I
          NORMAL RETURN
```

2. PURPOSE: Initialize the parameter J with the input integer.

CALLING SEQUENCES:

```
MAD       EXECUTE NDRN1A.(I)
FORTRAN   CALL NDRN1A(K)
UMAP      CALL    NDRN1A
          PAR     I
          NORMAL RETURN
```

ARGUMENTS:

I	Full word integer which may be used as an initialization of the parameter J. The initial normal value of I is 34359738367.
K	Name of an integer array of length 3. FORTRAN integers

NORMALLY DISTRIBUTED RANDOM NUMBER GENERATOR (CONTINUED)

are stored in the decrement portion of the machine word and hence the parameter J must be fed to NDRN1 in 3 parts. The initial normal value of K is K(1) = K(2) = 32767, K(3) = 31.

SUBROUTINES REQUIRED: ELOG, SQRT

SUBTRACE ON-OFF SWITCH

ENTRY POINTS: OFFTRC, ONTRC

PURPOSE: To provide control at execution time of the subroutine tracing feature of the system initiated by the $SUBTRACE processing function. These subroutines are effective only if the processing function was used. Executing OFFTRC will suppress the printing of the subroutine trace, and executing ONTRC will resume the printing if it has been suppressed.

CALLING SEQUENCES:

```
    MAD       EXECUTE OFFTRC.
              EXECUTE ONTRC.
    FORTRAN   CALL OFFTRC
              CALL ONTRC
    UMAP      CALL  OFFTRC
              CALL  ONTRC
```

PROGRAM COMMON PUNCH

ENTRY POINTS: PCPCH

PURPOSE: Produces absolute column binary cards with loading address, check sum, and ID.

CALLING SEQUENCES:

```
    MAD       EXECUTE PCPCH.(A,L1,L2,.......)
    UMAP      CALL    PCPCH
              PAR     A
              ...     L1
              ...     L2
              ...
              ...
              ...     LN
              NORMAL RETURN
```

ARGUMENTS:

A	First word of 2 BCD words to be used as ID on each card. Words stored in A, A-1
LI	Standard MAD-UMAP argument list elements.

SUBROUTINES REQUIRED: SPUNCH

PLOTTING SUBROUTINE

ENTRY POINTS: PLOT1, PLOT2, PLOT3, PLOT4, OMIT, FPLOT4

PURPOSE: Rapid machine plotting of numeric information for use with MAD, UMAP or FORTRAN calling programs. The resulting graph is copied onto a decimal output tape (6 for the MAMOS system) for subsequent off-line printing.

METHOD: The philosophy used in writing this routine was to treat a region of core storage (subsequently called the image region or simply the image) much as a piece of graph paper when plotting data manually.

First the image region is blanked out and a grid formed of I's and -'s (with +'s at the intersection points) is placed in the image. Given the numerical limits of the abscissa and ordinate (i.e., the maximum and minimum values of the two variables, say X and Y), the routine can place any specified BCD plotting character at the appropriate position in the image for a given pair of data values (X ,Y).

Each point (X ,Y) is plotted individually and independently of any preceding point. In other words, the data need not be presorted. Any number of points (X ,Y) with any corresponding BCD plotting characters can be placed in the image. A character falling on a grid line replaces the grid character in that position. A character falling on a previously plotted character will replace that character. Thus only the last plotted of two coincident data points appears in the final image. Points falling outside the grid limits (not in the image region) are ignored.

When all desired points have been placed in the region, the image is copied onto the specified decimal output tape for subsequent off-line printing or punching. Any number of duplicate copies of the graph can be produced.

USE: The subroutine has four entries which perform the following functions.

PLOT1 sets up the grid spacing and the total width and length of the graph image. It also determines the location of the decimal points and the multiplying factors (powers of 10) for values of the ordinate and abscissa to printed at the grid lines.

PLOT2 prepares the grid, examines the maximum and minimum values of the abscissa and ordinate and establishes internally a formula for computing the location in the image region corresponding to the point (X ,Y).

PLOT3 places a specified BCD plotting character in the appropriate position(s) corresponding to the given value(s) of (X ,Y).

PLOT4 (or FPLOT4) writes the image of the completed graph on the output tape for subsequent printing off line. A label for the ordinate is printed vertically (one character per line) at the left edge of the page. Values of the abscissa and ordinate are printed at the grid lines outside the bottom and left edges of the graph.

PLOTTING SUBROUTINE (CONTINUED)

+MAD CALLING SEQUENCES:

```
    EXECUTE PLOT1.(NSCALE, NHL, NSBH, NVL, NSBV)
    EXECUTE PLOT2.(IMAGE, XMAX, XMIN, YMAX, YMIN)
    EXECUTE PLOT3.(BCD, X, Y, NDATA)
    EXECUTE PLOT4.(NCHAR, LABEL)
```

+FORTRAN CALLING SEQUENCES:

```
    CALL PLOT1 (NSCALE, NHL, NSBH, NVL, NSBV)
    CALL PLOT2  (IMAGE, XMAX, XMIN, YMAX, YMIN)
    CALL PLOT3 (BCD, X, Y, NDATA)
    CALL FPLOT4 (NCHAR, NHABCDEF.....)
```

ARGUMENTS:

NSCALE Is a vector (array) in the users program having one or five locations. If the user wishes to use the standard scale factors and decimal point positions (see below), NSCALE should equal zero. To alter the standard factors NSCALE must be any non-zero quantity. In this case the NSCALE array must have five locations containing the following information.

FORTRAN *LOCATION	MAD LOCATION	CONTENTS	FUNCTION
NSCALE(1)	NSCALE(0)	**	Alter standard factors.
NSCALE(2)	NSCALE(1)	I	Printed values of the ordinate (Y) are 10.P.I times actual values.
NSCALE(3)	NSCALE(2)	J	Printed values of the ordinate (Y) have J digits following the decimal point (J.LE.8).
NSCALE(4)	NSCALE(3)	K	Printed values of the abscissa (X) are 10.P.K times actual values.
NSCALE(5)	NSCALE(4)	M	Printed values of the abscissa (X) have M digits following the decimal point (M.LE.9).

Standard scale factors. When NSCALE is zero the standard scale factors are used. The effective values of I, J, K, and M are 0, 3, 0, and 3 respectively. The actual values are printed with three decimal places.

NHL Is the number of horizontal grid lines in the graph image.
NSBH Is the number of spaces between horizontal grid lines.
NVL Is the number of vertical grid lines in the graph image.

\+ Restrictions and modes of arguments given later.
* The FORTRAN and MAD locations differ by one because FORTRAN arrays have no zeroth element.
** Any non-zero value.

PLOTTING SUBROUTINE (CONTINUED)

NSBV — Is the number of spaces between vertical grid lines.

NOTE: In keeping with standard notation for graph paper, (i.e., 10 X 10 to the inch) NHL and NVL are really one less than the actual number of lines. It is not customary to consider the axes when counting lines in the grid.

IMAGE — Is an array (vector), dimensioned in the users program, consisting of N sequential locations not used between execution of PLOT2 and PLOT4, where

N = P*(NSBH*NHL + 1)

P = (NSBV*NVL + 1)/6 rounded up to nearest integer.

XMAX — Is the value of the abscissa at the rightmost grid line.

XMIN — Is the value of the abscissa at the leftmost grid line.

YMAX — Is the value of the ordinate at the uppermost grid line.

YMIN — Is the value of the ordinate at the lowermost grid line.

BCD — Is the BCD (Hollerith) plotting character, and may be any legitimate left-adjusted BCD character (letter, digit, blank or special character *,.= etc.).

X — Is a single location (or array name) containing the X coordinate(s) of the point(s), (X ,Y).

Y — Is a single location (or array name) containing the Y coordinate(s) of the point(s), (X ,Y).

NDATA — Is the number of data points (X ,Y) associated with the arrays X and Y. With NDATA equal to 1, a single point will be plotted for a single execution of PLOT3. With NDATA equal to Q, Q points (X ,Y) taken in sequence from vectors of length Q starting at X and Y are plotted for a single execution of PLOT3.

NCHAR — Is the number of BCD (Hollerith) characters (including blanks) in the label array (vector).

LABEL — Is the name of an array (vector) which contains the string of BCD characters to be printed at the left edge of the output page, i.e., a label for the ordinate of the graph. This vector is stored backward in locations LABEL, LABEL-1, LABEL-2, ..., and in a MAD program will normally be preset with a vector values statement.

LABELLING THE ORDINATE: Use of FPLOT4 in FORTRAN.

As mentioned under the above explanation of the argument LABEL, the MAD user will normally preset the string of BCD characters to be printed along the left edge of the output page using a vector values statement. FORTRAN II, however, has no provision for presetting symbolic locations accessible to the programmer during execution. Consequently, the printing entry for FORTRAN calling programs, FPLOT4, has a somewhat different calling sequence than the MAD entry, PLOT4. The string of characters for the ordinate LABEL appears directly in the calling sequence as the second argument (Hollerith). The N preceding the H (specifying the Hollerith string) should be the same as the value of NCHAR.

PLOTTING SUBROUTINE (CONTINUED)

THE PLOTTING CHARACTER:

The plotting character can be set up by a substitution statement of the form;

```
BCD = 1H*                    (FORTRAN)
BCD = $*$                    (MAD)
```

or entered directly into the argument list for PLOT3 as;

```
CALL    PLOT3 (1H*,X,Y,NDATA)   (FORTRAN)
EXECUTE PLOT3.($*$,X,Y,NDATA)   (MAD)
```

UMAP CALLING SEQUENCES:

The UMAP calling sequences are identical with those compiled by MAD and FORTRAN. Examples are given in both the MAD manual and the FORTRAN II reference manual. If the label characters are preset in the normal UMAP fashion (ascending addresses in storage), the FPLOT4 entry should be used for printing.

```
CALL  FPLOT4
PAR   NCHAR
PAR   LABEL
```

In keeping with the SHARE-FORTRAN conventions, all index registers are preserved.

RESTRICTIONS ON ARGUMENTS:

NHL	.G.	C	
NSBH	.G.	C	
NVL	.G.	C	
NSBV	.G.	C	
(NSBV*NVL)	.LE.	113	
BCD			Must be a left-adjusted legitimate BCD (Hollerith) character, i.e. 1H-, 1H*, 1HA, 1H1, ETC.(FORTRAN) $-$, $*$, A, 1, ETC. (MAD)

MODES OF ARGUMENTS:

(1) Those arguments which deal directly with data values (XMAX, XMIN, YMAX, YMIN, X, Y) must be in floating point mode.

(2) Those arguments which deal with the arrangement of the image and the scale factors (NSCALE, NHL, NSBH, NVL, NSBV, NCHAR) and the number of data points NDATA can be;

- (A) Floating point.
- (B) FORTRAN type integers (15 binary bits in the decrement of the 7090/7094 machine word) of absolute value less than 32768.
- (c) MAD-like integers (low order 18 binary bits in the tag and address portions of the 7090/7094 machine word) of absolute value less than 262144.

The routine automatically determines which mode is being used for each argument.

(3) LABEL and BCD must contain Hollerith information only.

NOTE: The sign of NHL, NSBH, NVL, NSBV, NDATA, and NCHAR is ignored.

PLOTTING SUBROUTINE (CONTINUED)

SUGGESTIONS FOR THE USER:

1. Standard Grid.
   If the user desires to use a standard grid configuration with the standard scale factors, PLOT1 need not be executed. This standard graph consists of a full page graph 101 columns wide and 51 lines long with 10 vertical grid lines (NVL), 5 horizontal grid lines (NSBH and NSBV). The image array must be dimensioned at least 867 (decimal) locations.
2. Positioning the Graph on the Printed Page.
   The graph is always adjusted toward the left edge of the page. The topmost line is printed one space below the last line printed by the user before executing PLOT4. For example, to start at the top of a page the user can execute a statement of the type;

```
        PRINT COMMENT $1$          (MAD)
   or   WRITE OUTPUT TAPE 6, 5     (FORTRAN)
      5 FORMAT (1H1)
```

   prior to execution of PLOT4 or FPLOT4.
3. Printing Information Above and/or Below the Graph.
   If desired, the user can print a title above the graph before executing PLOT4 (or FPLOT4), or a label for the abscissa below the graph after executing PLOT4 (or FPLOT4). This is done, of course, with a PRINT FORMAT (MAD) or WRITE OUTPUT TAPE (FORTRAN) statement in the calling program.
4. Length of the LABEL Vector.
   The LABEL vector need not have as many characters as the number of printed lines in the final image. The characters given are printed in sequence, one per line, starting at the topmost line until all NCHAR characters have been used. Blanks are inserted automatically for any succeeding lines.
   If no LABEL for the ordinate is desired, set NCHAR to 0 (zero). The LABEL argument is then immaterial, but some second argument must be given for PLOT4 (or FPLOT4).
5. Plotting More Than One Set of Data - Changing the Plotting Character.
   The number of individual (or sets of) data points to be plotted is not limited in any way. PLOT3 may be executed as many times as desired. The plotting character can be the same or different for each of the PLOT3 executions.
6. Points Which Are Not Plotted.
   Data points will not be plotted under the following circumstances:
   1. The value of one or both of the coordinates (X,Y) lies outside the range XMIN to XMAX or YMIN to YMAX.
   2. The value of X or Y is not in floating point mode.
   3. NDATA = 0.

   Points which are not plotted are simply ignored by the routine. Thus, the user need not pretest his data for occurrence inside the grid limits.
7. Printing More Than One Copy of the Graph.
   PLOT4 (or FPLOT4) can be executed as many times as desired. One copy of the graph is produced per execution.

PLOTTING SUBROUTINE (CONTINUED)

8. Executing Other Instructions Between PLOT Entries.
   Any number of instructions can be executed between execution of successive PLOT entries, provided only that the image region is not disturbed between execution of PLOT2 and the final execution of PLOT4 (or FPLOT4). For example, if IMAGE is in common or erasable, no subroutines which use the same locations in common or erasable should be called between execution of PLOT2 and PLOT4 (or FPLOT4).
9. Graphs Which Cover More Than One Page.
   The number of lines in the graph is (NSBH*NHL+1). There is no limit on this quantity provided only that it is compatible with the number of locations in IMAGE (N on page 3). Thus a graph can cover anywhere from a small part of one page to several pages. For the multi-page graph, the normal skip between pages (across the page perforations) will take place. Where storage is a limiting factor, see section H for producing a graph of any arbitrary number of pages.

SPECIAL FEATURES FOR UNUSUAL APPLICATIONS:

A. Intermediate Printing While Plotting.
   After PLOT2 has prepared the grid and PLOT3 has plotted some data into the IMAGE region, PLOT4 (or FPLOT4) can be executed to give a current copy of the graph. PLOT3 can then be reexecuted to plot some more data. When PLOT4 (or FPLOT4) is reexecuted, the graph will contain all the points placed since the last PLOT2 entry.
B. Preparing a New Grid After Printing.
   If, after printing via PLOT4 (or FPLOT4), it is desired to prepare a new image of the same grid configuration as the previous one, it is only necessary to reexecute PLOT2, i.e., PLOT1 need not be reexecuted if the arguments would be the same as used for the previous execution of PLOT1.
C. Printing Numeric Fields As Integers (Without Decimal Point).
   If integer printout of the numeric values of the abscissa or ordinate at the grid lines is desired, the appropriate decimal point parameter in the NSCALE array should be set to any negative value less than or equal to -1.
D. Printing In the Body of the Graph.
   Since any BCD characters can be placed in the IMAGE, given the proper coordinates, a title can be placed in the body of the IMAGE by plotting one character at a time with the appropriate coordinates. However, since the entire IMAGE region is available at all times, alphabetic constants (6 BCD characters per 7090/7094 word) can be placed directly into the image using a substitution statement of the type,

```
IMAGE(M) = $Y VS X$        (MAD)
```

   where M is the appropriate subscript in the IMAGE vector. To determine the proper M, a short description of the image region layout follows. Each horizontal line image for the printed page uses P locations in the IMAGE array where P is calculated as previously described in this writeup (see section on arguments). For a MAD program, the first (top) line is assigned to locations IMAGE(0)...IMAGE(P-1), the second line from IMAGE(P)...IMAGE(2*P-1), the third from IMAGE(2*P)... IMAGE(3*P-1), etc. If it was desired to print ' Y VERSUS X ' in the

PLOTTING SUBROUTINE (CONTINUED)

center of the third line of the standard IMAGE, for which P = 17, the statements

```
IMAGE(42) = $Y  VER$      (MAD)
IMAGE(43) = $SUS  X$
```

could be executed after execution of PLOT2 and before execution of PLOT4. For FORTRAN programs (which have no zeroth subscript), these subscripts would be one greater in all cases, i.e., the first line would be assigned to locations IMAGE(1)...IMAGE(P), etc. The substitution statements would be of the form

```
GRAPH(43) = 6HY  VER      (FORTRAN)
GRAPH(44) = 6HSUS  X
```

NOTE: Because FORTRAN II restricts integers to the decrement of the 7090/7094 machine word, substitution statements involving BCD constants can be used only if the variable involved has a floating point name. Hence, the use of the name 'GRAPH' instead of 'IMAGE' for the IMAGE region. Since this type of substitution statement is not a recognized part of the FORTRAN language, but is nevertheless accepted by the 7090/7094 FORTRAN II compiler, care should be exercized in its use.

E. Modification of the Grid System.
As written, the routine prepares a Cartesian grid system only. However, by setting NHL and NVL both equal to 1 (one), only the border lines of the IMAGE will be prepared by PLOT2. The user can subsequently prepare his own grid (log-scaled for example) by plotting it (the grid) with repeated execution of PLOT3 using the desired grid character as the plotting character BCD and the appropriate coordinates, or grid lines can be laid down using the technique of D, above.

F. Blanking Out the Grid System or Undesired Character(s).
If the user desires no grid at all (including the borders) he can blank out the entire IMAGE array after execution of PLOT2 but before execution of PLOT3. When PLOT4 (or FPLOT4) is subsequently executed, values of the ordinate and abscissa will be printed at the borders as if the blanking operation had never taken place (unless the subroutine OMIT has been executed, see G below).

G. Deleting the Printing of Certain Portions of the Graph.
Provision has been made for deleting the printout of the following items:

1. Numeric values of the abscissa at the grid lines.
2. Numeric values of the ordinate at the grid lines.
3. Items (1) and (2).
4. The complete bottom horizontal grid line.
5. Items (1) and (4).
6. Items (2) and (4).
7. Items (1), (2) and (4).

This is accomplished by executing the entry OMIT any time before execution of PLOT4 (or FPLOT4). The calling sequence for OMIT is;

```
EXECUTE OMIT.(ARG)      (MAD)
or      CALL    OMIT (ARG)      (FORTRAN)
```

where ARG is a positive number corresponding to one of the above

PLOTTING SUBROUTINE (CONTINUED)

seven numbers. If ARG is greater than 7, it will be treated modulo 8. ARG may be of any mode.

To restore printing of any of the seven items, OMIT can be called with ARG a negative number corresponding to the number of the item(s) to be restored.

Examples where it would be desirable to delete printout of the ordinate and/or abscissa values might be the plotting of a histogram or when the grid system has been modified (see E and F, above). The feature of deleting the printing of the bottom line is useful for joining two graph segments (see H, below).

NOTE: Deleting printout of the bottom line by EXECUTE OMIT.(4) does not delete printout of the abscissa values. The procedure for deleting both the bottom line and the abscissa values is EXECUTE OMIT.(5).

H. Producing a Graph of Any Arbitrary Size.

Because of storage limitations, there is a practical upper limit to the size of the IMAGE region, and, hence, to the size of the graph which can be produced by a single execution of the sequence PLOT2, PLOT3, PLOT4. However, a graph of any arbitrary size can be prepared in piecewise fashion where the IMAGE region at any one time contains only one segment of the complete graph. As each segment is prepared by PLOT2 (with its appropriate YMAX and YMIN) and printed (with the bottom line deleted, see G, above) by PLOT4 (or FPLOT4), it will join with the previous segment to form the appearance of one continuous graph. The following example written in MAD illustrates the preparation of a graph of N segments.

```
          .
          .
          EXECUTE OMIT.(5)
          PRINT FORMAT TOP
          EXECUTE PLOT1.(0,1,12,4,25)
          DELTAY = (YMAX-YMIN)/N
          THROUGH LAST, FOR I = C,1,I.E.N
          EXECUTE PLOT2.(IMAGE,XMAX,XMIN,YMAX-I*DELTAY,
         1  YMAX-(I+1)*DELTAY)
          EXECUTE PLOT3.($*$,X,Y,NDATA)
          WHENEVER I.E.N-1, EXECUTE OMIT.(-5)
LAST      EXECUTE PLOT4.(12,LABEL(2*I))
          PRINT FORMAT BOTTOM
          .
          .
          VECTOR VALUES TOP = $ . . . .
          VECTOR VALUES BOTTOM = $ . . . .
          VECTOR VALUES LABEL = $ . . . .
          INTEGER I, N
          DIMENSION IMAGE(221)
```

NOTE: If this technique is used, only one complete copy of the graph can be produced.

PLOTTING SUBROUTINE (CONTINUED)

ERROR CHECKS:

PRINTED COMMENTS:

The subroutine contains many error checking features. For arguments incompatible with the restrictions listed previously, the comment "IMPROPER ARGUMENT" will be printed along with the appropriate entry PLOT1, PLOT2, etc.

If an error occurs in PLOT1 and/or PLOT2, the comment will be printed and subsequent executions of later entries will be deleted without comment (see computation switch below), until the offending entry is executed successfully.

If the user attempts to execute PLOT3, PLOT4 or FPLOT4 without a previous execution of PLOT2 (or, without execution of PLOT2, subsequent to any execution of PLOT1), the comment "NO PREVIOUS PLOT2" will be printed.

COMPUTATION SWITCH:

A floating point constant is returned to the accumulator after execution of each entry. If no difficulties are encountered during execution, a zero is returned. An error in PLOT1, PLOT2, PLOT3 or PLOT4 (or FPLOT4), or a deletion of execution caused by the unsuccessful execution of an earlier entry, causes a 1.0, 2.0, 3.0 or 4.0, respectively, to be returned. The calling sequences for any or all MAD entries can thus be altered to form

```
R = PLOT2.(IMAGE,XMAX,XMIN,YMAX,YMIN)        (MAD)
WHENEVER R.G.0, TRANSFER TO TRUBL
R = PLOT3.($*$,.....
TRUBL  (The error condition statement)
```

Because FORTRAN does not permit the use of Hollerith arguments in the calling sequence of functions (appearing in expressions), the FORTRAN user has access to the computation switch from PLOT3 only if the plotting character is entered as a variable name rather than as a Hollerith argument. Thus

```
R = PLOT3(BCD,X,Y,NDATA) is legal
while   R = PLOT3(1H*,X,Y,NDATA) is not legal.
```

By the same token, the switch is not available for FPLOT4 when it is called with a Hollerith argument (when a label for the ordinate is desired). However, the switch is still available when FPLOT4 is called by a UMAP program.

A typical FORTRAN sequence using the switches might appear as follows:

```
        BCD = 1H*
        R = PLOT1(NSCALE,NHL,NSBH,NVL,NSBV)        (FORTRAN)
        IF (R) 102,102,110
    102 R = PLOT2(IMAGE, XMAX,XMIN,YMAX,YMIN)
        IF (R) 103,103,110
    103 R = PLOT3(BCD,X,Y,NDATA)
        IF (R) 104,104,110
    104 CALL FPLOT4 (.............
    110 (The error condition statement
```

If any points are not plotted by PLOT3, a -3.0 is returned to the accumulator. This may or may not be considered an error.

PLOTTING SUBROUTINE (CONTINUED)

ARGUMENT MODIFICATION BY THE ROUTINE:

In some cases, the routine modifies the positioning of the decimal point or shifts the entire abscissa printout to accomodate all the desired numbers in the width of the printed page.

If the suggested scale factors are such that overflow or underflow of the machine word would result, the factor is reset to 0.

If the value of the ordinate or abscissa is too large to be printed in the alloted space, it will be truncated from the left in printout.

Under no circumstances is the content of an argument location in the calling program modified.

SUBROUTINES REQUIRED: SPRINT

SAMPLE PROBLEMS FOR THE PLOTTING SUBROUTINE:

Three sample problems have been prepared. The first has been coded in both MAD and FORTRAN, the second two have been coded in MAD only.

The first is a simple example which reads a set of data values from input cards and plots them, illustrating;

(1) Use of standard grid and scale factors
(2) Plotting of an array of data points with a single execution of PLOT3.

The second is a more complex example which plots the solutions Y and DY/DX to the differential equation

D2Y/DX2 + A DY/DX + B = 0.

The differential equation is solved using the Runge-Kutta subroutines from the library tape (see RKDEQ,SETRKD write-ups.) It illustrates:

(1) Modification of standard grid and scale factors.
(2) Plotting of one character per execution of PLOT3.
(3) Use of different plotting characters in the same graph.
(4) Printing a variable number of copies of the graph.

The third prepares a polar plot of the polar function Polf, a spiral of the form

R = K*ANGLE.

This program illustrates:

(1) Erasure of the grid prepared by PLOT2.
(2) Placing a new grid in the IMAGE region by plotting the grid characters one at a time use PLOT3.
(3) Printing inside the graph by placing BCD constants in the image region.
(4) Use of the computation switch to stop the iteration loop involving the plotting of data points.
(5) Use of the subroutine OMIT to delete the printing of numbers at the edge of the graph.

All examples illustrate the printing of a label for the ordinate and the first two show the printing of a title above the graph and a label for the abscissa below the graph.

PLOTTING SUBROUTINE (CONTINUED)

SAMPLE PROBLEM NUMBER ONE.

```
$COMPILE MAD, PUNCH OBJECT                                          PLMADOOO
          R
          R PROGRAM TO ILLUSTRATE PLOTTING MULTIPLE POINTS WITH MAD
          R      USING THE STANDARD GRID AND SCALE FACTORS
          R
           DIMENSION X(100), Y(100), GRAPH(867)
           INTEGER N
FIRST      READ FORMAT ENTR, N,XMAX,XMIN,YMAX,YMIN
           READ FORMAT DATA, X(1)...X(N)
           READ FORMAT DATA, Y(1)...Y(N)
           EXECUTE PLOT2.(GRAPH,XMAX,XMIN,YMAX,YMIN)
           EXECUTE PLOT3.($*$,X(1),Y(1),N)
           PRINT FORMAT TITLE
           EXECUTE PLOT4.(32,ORD)
           PRINT FORMAT ABS
           TRANSFER TO FIRST
          R
          R FORMAT STATEMENTS
          R
           VECTOR VALUES ENTR = $I10,4F10.4*$
           VECTOR VALUES DATA =$7F10.4*$
           VECTOR VALUES TITLE =$1H1,S54,14HPLOT OF X VS Y   /1H *$
           VECTOR VALUES ABS = $1H0,S55,14HTHE ABSCISSA X *$
           VECTOR VALUES ORD =$                  THE ORDINATE Y $
           END OF PROGRAM
$DATA
```

```
$MADTRAN,PRINT OBJECT, PUNCH OBJECT                                 PLFTROOO
C
C     PROGRAM TO ILLUSTRATE PLOTTING MULTIPLE POINTS WITH FORTRAN
C          USING THE STANDARD GRID AND SCALE FACTORS
C
      DIMENSION X(100),Y(100),GRAPH(867)
    1 READ INPUT TAPE 7,100, N, XMAX, XMIN, YMAX, YMIN
      READ INPUT TAPE 7,101,(X(I), I=1,N)
      READ INPUT TAPE 7,101,(Y(I), I=1,N)
      CALL PLOT2(GRAPH, XMAX,XMIN,YMAX,YMIN)
      CALL PLOT3(1H*,X(1),Y(1),N)
      WRITE OUTPUT TAPE 6,102
      CALL PLOT4(32, 32H                THE ORDINATE Y )
      WRITE OUTPUT TAPE 6,103
      GO TO 1
C
C     FORMAT STATEMENTS
C
  100 FORMAT (I10,4F10.4)
  101 FORMAT (7F10.9)
  102 FORMAT (1H1, 54X,    14HPLOT OF X VS Y   /1H )
  103 FORMAT (1H0, 55X,    14HTHE ABSCISSA X )
      END
$DATA
```

PLOTTING SUBROUTINE (CONTINUED)

SAMPLE PROBLEM NUMBER TWO

```
$COMPILE MAD, PUNCH OBJECT                                          RKMADOOO
          R
          R  PROGRAM TO ILLUSTRATE PLOTTING ONE CHARACTER AT A TIME
          R  WITH MAD
          R
           DIMENSION F(2),Q(2),Z(2),DUMMY(833)
           INTEGER K,NPLOTS
           VECTOR VALUES MARGIN = $                    Y AND PRIME$
           VECTOR VALUES N = 1,0,1,0,1
           EXECUTE SETRKD.(2,Z(1),F(1),Q,X,H)
FIRST      READ FORMAT INPUT,A,B,Z(1),Z(2),H,MAXY,MINY,NPLOTS
           EXECUTE PLOT1.(N,4,12,5,20)
           EXECUTE PLOT2.(DUMMY,10.,0.,MAXY,MINY)
           PRINT FORMAT HEAD
           X = 0.
WRITE      PRINT FORMAT RESULT,X,Z(1),Z(2)
           EXECUTE PLOT3.($*$,X ,Z(1),1)
           EXECUTE PLOT3.($+$,X ,Z(2),1)
           WHENEVER X.G.10.,TRANSFER TO OUT
STEP       K = RKDEQ.(0)
           WHENEVER K.E.1
              F(1) = Z(2)
              F(2) = -A*Z(2) - B*Z(1)
              TRANSFER TO STEP
           END OF CONDITIONAL
           TRANSFER TO WRITE
OUT        THROUGH LOOP,FOR K = 1,1,K.G.NPLOTS
           PRINT FORMAT TITLE,A,B
           EXECUTE PLOT4.(30,MARGIN)
LOOP       PRINT FORMAT BOTTOM
           TRANSFER TO FIRST
          R
          R   FORMAT STATEMENTS
          R
           VECTOR VALUES INPUT = $7F10.5,I2*$
           VECTOR VALUES HEAD = $1H1,S13,47HTABULATED SOLUTION OF THE DIF
          1FERENTIAL EQUATION /1HO,S17,1HX,S19,1HY,S16,6HYPRIME*$
           VECTOR VALUES RESULT = $1H ,3F20.4*$
           VECTOR VALUES TITLE = $1H1,S43, 37HSOLUTION OF THE DIFFERENTIA
          1L EQUATION /1H ,S43,12HY-DOT-DOT + F5.2,9H Y-DOT + F5.2,
          26H Y = 0 /1HO*$
           VECTOR VALUES BOTTOM=$1HO,S49,26HTHE INDEPENDENT VARIABLE X//
          11H S44, 36HPLOTTING CHARACTERS. Y(*), YPRIME(+) *$
           END OF PROGRAM
```

PLOTTING SUBROUTINE (CONTINUED)

SAMPLE PROBLEM NUMBER THREE.

```
$COMPILE MAD, EXECUTE, DUMP, PUNCH OBJECT                          POLAROOO
          R
          R  POLAR PLOT OF A SPIRAL TO ILLUSTRATE ERASURE OF THE GRID,
          R  PREPARATION OF A NEW GRID, PRINTING INSIDE THE IMAGE,
          R  USE OF THE SUBROUTINE OMIT. AND THE COMPUTATION SWITCH.
          R
           INTERNAL FUNCTION POLF.(ANGLE) = K*ANGLE
           DIMENSION IMAGE(686)
           BOOLEAN BOOL
           INTEGER IMAGE, CHAR, I
START      READ FORMAT INPUT , RMAX
           EXECUTE PLOT1. (0,1,48,1,80)
           EXECUTE PLOT2.(IMAGE,RMAX,-RMAX,RMAX,-RMAX)
           THROUGH ERASE, FOR I = 0,1, I.G.686
ERASE      IMAGE(I) = $       $
           DELTAY = RMAX/24.
           DELTAX = RMAX/40.
           THROUGH PLACI, FOR Y=RMAX,-DELTAY,Y.L.-RMAX
PLACI      EXECUTE PLOT3.($I$,0.0,Y,1)
           THROUGH PLACP,  FOR Y=RMAX,-1.0,Y.L.-RMAX
PLACP      EXECUTE PLOT3.($+$,0.0,Y,1)
           THROUGH PLACM1, FOR X =-RMAX,DELTAX,X.G.RMAX
PLACM1     EXECUTE PLOT3.($-$,X,0.0,1)
           THROUGH PLACPL, FOR X =-RMAX,1.0,X.G.RMAX
PLACPL     EXECUTE PLOT3.($+$,X,0.0,1)
           IMAGE(349) = $- 0   $
           IMAGE(6) = $   90 $
           IMAGE(336) = $180 --$
           IMAGE(678) = $   270$
READ       READ FORMAT DATA, CHAR, K, BOOL
           SWITCH = 0.
           THROUGH PLACE, FOR THET = 0., .1, THET.G.25..OR.SWITCH.NE.0.
PLACE      SWITCH=PLOT3.(CHAR,POLF.(THET)*COS.(THET),POLF.(THET)*
          1SIN.(THET),1)
           PRINT FORMAT SKIP
           EXECUTE PLOT4.(36,MARGIN)
           WHENEVER BOOL, TRANSFER TO START
           TRANSFER TO READ
           VECTOR VALUES INPUT = $F10.4*$
           VECTOR VALUES DATA = $C1,F9.4,I1*$
           VECTOR VALUES SKIP = $1H1/1HO/1HO*$
           VECTOR VALUES MARGIN = $            POLAR PLOT OF THE SPIRAL$
           END OF PROGRAM
$DATA
4.0
* -.25
```

UNIFORMLY DISTRIBUTED RANDOM NUMBER GENERATOR

ENTRY POINTS: RAM2A, RAM2B, RAM2C, RAM2D

PURPOSE: Produce a random number in the interval (0,1). A set of random numbers generated by RAM2 has a uniform distribution.

CALLING SEQUENCES:

```
MAD       X = RAM2B.(0)
FORTRAN   X = RAM2B(0)
UMAP      CALL    RAM2B
          PAR     0
          NORMAL RETURN - X IN THE ACCUMULATOR
```

ARGUMENTS:

X Floating point random number, 0 .LE. X .LE. 1.

SPECIAL ENTRIES:

RAM2B uses an integer parameter J to compute each random number. J is initially set at 2.P.35 - 1 and changes with each execution of RAM2B. The following special entries enable the user to pick up the current value of J and to use it as input to RAM2 in order to initialize a sequence of random numbers at a later execution of the calling program.

1. PURPOSE: Save the current value of J that would have been used to calculate the next random number.
   CALLING SEQUENCES:

```
MAD       I = RAM2D.(0)   (INTEGER MODE)
FORTRAN   CALL RAM2C(K)
UMAP      CALL    RAM2D
          PAR     I
          NORMAL RETURN
```

2. PURPOSE: Initialize the parameter J with the input integer.
   CALLING SEQUENCES:

```
MAD       EXECUTE RAM2A.(I)
FORTRAN   CALL RAM2A(K)
UMAP      CALL    RAM2A
          PAR     I
          NORMAL RETURN
```

ARGUMENTS:

I Full word integer which may be used as an initialization of the parameter J. The initial normal value of I is 34359738367.

K Name of an integer array of length 3. FORTRAN integers are stored in the decrement portion of the machine word and, hence, the parameter J must be fed to RAM2 in 3 parts. The initial normal value of K is K(1) = K(2) = 32767, K(3) = 31.

UNIFORMLY DISTRIBUTED RANDOM NUMBER GENERATOR

ENTRY POINTS: RANDOM

PURPOSE: To provide the means for generating random numbers, uniformly distributed over the interval 0.LE.X.LE.1.

CALLING SEQUENCES:

MAD	Y = RANDOM.(RNO)
FORTRAN	Y = RANDOM (RNO)
UMAP	CALL RANDOM,RNO
	STO Y

ARGUMENTS:

Y — The variable whose value is set by the subroutine RANDOM.

RNO — The variable whose value is used to propagate the generation of random numbers. The initial value of RNO may be set by the user by reading into the location. Any non-zero positive initial value will cause the routine to generate the pseudo-random sequence corresponding to the initial value. If the initial value of RNO is zero, the routine will perform a logical checksum of certain locations in low core, including the clock and various buffers, so that the probability of repeating the same sequence of random numbers on successive approaches to the machine is very small (less than .0000002). The argument RNO must not be an absolute constant. For example, the calling sequence Y = RANDOM.(0) is completely incorrect and will result in completely meaningless operation of the program. (The subroutine RANDOM. modifies the value of the argument on each entry. Thus, in this illustration, the effect is to change the value of the constant zero to random values. If the modified value of the constant zero were used to clear an array, obvious chaos would result)

COMMENTS: RANDOM employs the thoroughly tested power residue method of random number generation. The periodicity of this method is 2.P.35 - 1.

NORMALLY DISTRIBUTED RANDOM NUMBER GENERATOR

ENTRY POINTS: RANDND

PURPOSE: To provide the means for generation of random numbers whose distribution has a given mean value and given standard deviation.

CALLING SEQUENCES:

MAD	Y = RANDND.(MEAN,SIGMA,RNO)
FORTRAN	Y = RANDND(XMEAN,SIGMA,RNO)
UMAP	CALL RANDND,MEAN,SIGMA,RNO

ARGUMENTS:

Y	The variable whose value is set by the subroutine RANDND.
MEAN (XMEAN)	The floating point variable whose value is the mean of the desired distribution.
SIGMA	The variable whose value is used to propagate the random number sequence. Since RANDND calls upon RANDOM, the user should read the write-up on the subroutine RANDOM.

SUBROUTINES REQUIRED: RANDOM, ELOG, SQRT.

REPLACE TAPES

ENTRY POINTS: REPLCE

PURPOSE: Prints instructions to the operator to replace a user's tapes with other tapes belonging to the user. It first calls on DISMNT and MOUNT and then stops for the operator to replace the tapes. Labels on the new tapes are handled in the same way as if they were mounted with MOUNT.

CALLING SEQUENCES:

```
MAD      DATE = REPLCE.(NUM1,NAME1,UNIT1,DENS1,PRO1,MODE1,NUM2,NAME2,
                  UNIT2,DENS2,PRO2,MODE2,...,NUMN,NAMEN,UNITN,DENSN,
                  PRON,MODEN)
FORTRAN  DATE = REPLCE(NUM1,NAME1,UNIT1,DENS1,PRO1,MODE1,NUM2,NAME2
                  UNIT2,DENS2,PRO2,MODE2,...,NUMN,NAMEN,UNITN,DENSN,
                  PRON,MODEN)

UMAP     CALL     REPLCE
         PAR      NUM1
         PAR      NAME1
         PAR      UNIT1
         PAR      DENS1
         PAR      PRO1
         PAR      MODE1
         ...      ...
         PAR      NUMN
         PAR      NAMEN
         PAR      UNITN
         PAR      DENSN
         PAR      PRON
         PAR      MODEN

         DATE IS RETURNED IN THE ACCUMULATOR.
```

ARGUMENTS:

There are six arguments for each tape to be replaced.

NUM	Number (full word or FORTRAN integer) assigned to the new tape. See MOUNT for use of zero and negative numbers.
NAME	One word BCD name used in tape label of new tape.
UNIT	Logical number (full word or FORTRAN integer) of the tape drive of the tape to be replaced. If this number is positive, the ring of the replaced tape will be removed. If unit is negative, a ring will be inserted in the replaced tape.
DENS	If zero, the new tape will be in low density. If non-zero, the new tape will be in high density. (This should normally be non-zero.)
PRO	If zero, file protect ring should be inserted in the new tape. If non-zero, file protect ring should be removed from the new tape.

MODE — If zero, the new tape will be BCD. If non-zero, the new tape will be binary.

DATE — The BCD date in the label of the new tape. It has the form DD/MM where DD is the day of the month MM. If there is more than one group of arguments, the date will be that on the tape specified by the last group.

SUBROUTINES REQUIRED: DISMNT, MOUNT

RUNGE-KUTTA SOLUTION OF DIFFERENTIAL EQUATIONS

ENTRY POINTS: RKDEQ, SETRKD

PURPOSE: Solves a system of N first order ordinary differential equations by the Runge-Kutta fourth-order method. The equations are assumed to be of the form:

```
DY(1)/DX = F(1)(X,Y(1),...,Y(N))
   ..          .   .   .
   ..          .   .   .
DY(N)/DX = F(N)(X,Y(1),...,Y(N))
```

where X is the independent variable.

CALLING SEQUENCES:

```
MAD       EXECUTE SETRKD.(N,Y,F,Q,X,H)          SETUP
          S = RKDEQ.(0)                         EXECUTION

FORTRAN   CALL SETRKD (N,Y,F,Q,X,H)             SETUP
          S = RKDEQ(0)                          EXECUTION
```

```
UMAP            SETUP              EXECUTION
          CALL      SETRKD         CALL    RKDEQ
          PAR       N              NORMAL RETURN - S IN THE
          PAR       Y              ACCUMULATOR
          PAR       F
          PAR       Q
          PAR       X
          PAR       H
          NORMAL RETURN
```

SETRKD. must be entered before the first entry to RKDEQ. Thereafter, SETRKD. must be entered only when a change occurs in the parameter list. (Note that this does not include changes in parameter values.)

NOTE: In MAD and FORTRAN, the arguments used in SETRKD must be single variable names, subscripted if desired, except for N and H, which may be constants. Expressions may not be used as arguments.

ARGUMENTS:

N Number of equations to be solved (integer variable or constant).

Y Name of the first element of a floating point vector in which the solution values Y(I) will be stored by RKDEQ. The initial values of Y(I) should be stored here prior to the first entry.

F Name of the first element of a floating point vector in which the values F(I) of the derivatives are stored.

Q Name of the first element of a temporary storage region used by RKDEQ. This region must be of length at least N.

X The floating point value of the independent variable X. This must be set to the initial value of X prior to the first entry. The independent variable is automatically incremented by RKDEQ.

H Floating point value of the increment for X, i.e., the step size. This value may be changed between solution points, if desired.

S Floating point computation switch,
S = 1.0 compute the values of the derivatives F(I) using the current values of X and Y(I) and return to RKDEQ.
S = 2.0 the solution values Y(I) for the present value of X are stored in Y.

SAVE BLOCKS OF CORE FOR LATER RELOADING BY SYSTEM

ENTRY POINTS: SAVCOR

PURPOSE: Saves up to ten blocks of core on designated tape as a standard binary system record, with checksum, for later loading by SEQPGM (if tape is properly positioned in front of the record first), or by SELRCD. After reloading, either of 2 alternate exits may be taken.

CALLING SEQUENCE:

```
MAD     EXECUTE SAVCOR.(TAPE,A1,...,B1,A2,...,B2, ETC., AN,...,BN,
                        EXIT1,EXIT2)

UMAP    CALL    SAVCOR
        PAR     TAPE
        BLK     A1,,B1
        BLK     A2,,B2
        ...      ...
        ...      ...
        BLK     AN,,BN
        PAR     EXIT1
        PAR     EXIT2
```

ARGUMENTS:

TAPE	Logical tape number of tape on which core is to be saved. Tape is not positioned before or after writing, nor is an end-of-file written before or after writing.
A1,B1	Blocks of core locations from A1 to B1,
A2,B2	A2 to B2, etc., are written as one record.
...	when later reloaded, these blocks will load
AN,BN	into original locations.
EXIT1	Location to which transfer is to be made if sense light 2 is off when record is reloaded.
EXIT2	Location to which transfer is to be made if sense light 2 is on when record is reloaded.

RESTRICTION: A maximum of ten blocks may be used.

SUBROUTINES USED: WRSBIN

CALLING SUBROUTINES FOR PING-PONG SEGMENTS

ENTRY POINTS: SELPGM, SEQPGM

PURPOSE: SEQPGM is used in ping-pong to call the next core in sequence. SELPGM is used to select one of the cores as the next core to be executed.

CALLING SEQUENCES:

MAD
```
EXECUTE SEQPGM.(TAPE)
EXECUTE SELPGM.(I,TAPE)
```

FORTRAN
```
CALL SEQPGM(TAPE)
CALL SELPGM(I,TAPE)
```

UMAP
```
CALL     SEQPGM
PAR      TAPE
NO RETURN

CALL     SELPGM
PAR      I
PAR      TAPE
NO RETURN
```

ARGUMENTS:

I	Integer core number designated to be the next core to be executed.
TAPE	Tape on which the next core will be found. This parameter may be omitted. If omitted, tape 2 will be assumed.

SUBROUTINES REQUIRED: SELRCD

SET LOW CORE TRAP LOCATIONS

ENTRY POINTS: SET2, SET8

PURPOSE: To allow users to set trap return locations in low core with their own transfers.
SET2 sets location 2, which is the trap location for the STR instruction.
SET8 sets location 8, which is the trap location for floating-point trap.

NOTE: The I/O routines and the list manipulation routines (.SET, .SAVE, .RSTOR) use STR's and hence location 2. They do not save the previous instruction that was in the location. Also, during I/O and LIST manipulation, the user must not modify location 2 from the TSX to the subroutine until after the ENDIO (STR 0).

CALLING SEQUENCES:

MAD
```
EXECUTE SET2.(L)
EXECUTE SET8.(L)
```

FORTRAN
```
CALL SET2(M)
CALL SET8(M)
```
See point 7 under arguments in introduction to this section.

UMAP
```
CALL SET2          CALL SET8
PAR  X             PAR  X
```

ARGUMENTS:

L Statement label of the statement to go to when the trap occurs.

M A variable that has been assigned the formula number of the statement to go to when the trap occurs.

X A location containing 'TTR RETURN' where return is the location to go to when the trap occurs.

SET END OF FILE RETURN

ENTRY POINTS: SETEOF, SETEFL

PURPOSE: Normally, when an end-of-file is found on a tape while reading, if the tape is the input tape, the comment '**** ALL INPUT DATA HAVE BEEN PROCESSED' is printed and the job is terminated. If the tape is a scratch tape, the comment '**** END OF FILE ON SCRATCH TAPE' is printed and the job is terminated. This procedure can be altered by using SETEOF or SETEFL.

CALLING SEQUENCES:

MAD
```
EXECUTE SETEOF. (LOC,N)
EXECUTE SETEOF. (0)
```

FORTRAN
```
CALL SETEFL (LOC)
CALL SETEFL (0)
```

UMAP
```
CALL SETEOF          CALL SETEOF
PAR  LOC             PAR  ZERO
PAR  N
```

ARGUMENTS:

LOC Instruction to which control is transferred when an end of file mark is encountered by the input read subroutines. (FORTRAN users should see point 7 under arguments in the introduction to this write-up.)

0 This is the integer zero. After it is used as the argument, an end of file mark will cause a normal return to the executive system.

N This argument is optional. If used, N will be set equal to the logical number (integer mode) of the tape being read.

ZERO A location containing zero.

SET I/O ERROR RETURN

ENTRY POINTS: SETERR, .ERR

PURPOSE: Allows user to retain control when an error is detected by the input-output subroutines. Otherwise, control is returned to the executive system, which terminates the job.

CALLING SEQUENCES:

MAD
```
EXECUTE SETERR.(LOC,E)
EXECUTE SETERR.(0)
```

FORTRAN
```
CALL SETERR(LOC)
CALL SETERR (0)
```

UMAP
```
CALL SETERR    CALL SETERR
PAR  LOC       PAR  ZERO
PAR  E
```

ARGUMENTS:

LOC Instruction to which control is transferred when an error is detected by the I/O subroutines. (FORTRAN users should see point 7 under arguments in the introduction to this write-up.)

0 This is the integer zero. After it is used as the argument, an error detected by the I/O subroutines will cause a normal return to the executive system.

E This argument is optional. If used, E will be set equal to the integer error number (see appendix I).

ZERO A location containing zero.

(.ERR is the entry point to this subroutine that the I/O routines use when an error has been found. They print out the error comment and then call on .ERR to either go to the user or return to the system.)

SUBROUTINES REQUIRED: ERROR, SPRINT, SYSTEM

SET END OF TAPE TEST OPTION

ENTRY POINTS: SETETT

PURPOSE: Allows user to change normal procedure when end of tape is encountered during a write operation (except on output tape).

CALLING SEQUENCES:

MAD
```
EXECUTE SETETT.(LOC,N)
EXECUTE SETETT.(0)
```

FORTRAN
```
CALL SETETT (LOC)
CALL SETETT (0)
```

UMAP
```
CALL SETETT          CALL SETETT
PAR  LOC             PAR  ZERO
PAR  N
```

ARGUMENTS:

LOC — Instruction to which control is transferred when end of tape is encountered while writing a tape other than the output tape. (FORTRAN users should see point 7 under arguments in the introduction to this section.)

0 — This is the integer zero. After it is used as the argument, an end-of-file mark is placed on the tape, and the tape is rewound and unloaded for replacement by the operator. This is the normal procedure.

N — This is an optional argument. If given, N will be set equal to the logical number of the tape on which the end of tape was encountered.

ZERO — A location containing zero.

FLOATING POINT TRAP CONTROL

ENTRY POINTS: SETFTP, RSTFTP

PURPOSE: SETFTP allows the user to supply his own trap control routine for floating point trapping.
RSTFTP restores trap control to the system trap routine.

CALLING SEQUENCES:

MAD
```
EXECUTE SETFTP.(WHERE,WHAT,WHRTO)
EXECUTE RSTFTP.
```

UMAP
```
CALL SETFTP,WHERE,WHAT,WHRTO
CALL RSTFTP
```

ARGUMENTS:

WHERE — Location where the floating trap occurred.

WHAT — Trap bit information from location 0.
Each of the four right-hand bits that is one has the meaning listed below.

BIT POSITION	MEANING
32	Operation was a divide.
33	Overflow in either AC or MQ or both.
34	AC factor exceeded.
35	MQ fraction is excessive.

(See IBM 7090/7094 Reference Manual)

WHRTO Location to go to when trap occurs.

SET UP FOR PLOT SUBROUTINE (IT MAKES PLOT PAINLESS)

ENTRY POINTS: SETPLT, USTPLT

PURPOSE: This subroutine is designed to be used with the PLOT subroutine (which is on library tape). The PLOT subroutine produces graphs of the quantities given it by the user. (For a detailed explanation, see the PLOT write-up.) It is a powerful and versatile tool, but is, as a result, rather complicated and clumsy to use. It requires that the user make 4 entries to the subroutine with a total of 16 arguments, and in order to determine the values for these arguments (such as the number of horizontal lines, number of spaces between horizontal lines, etc.), the user must do considerable precalculation. The user must also know the range of answers in advance so he can set the maximum and minimum values for the abscissa and for the ordinate. This is all work that can be done by the computer, and SETPLT is a subroutine that does it.

FEATURES: SETPLT inspects the data to be plotted, calculates the arguments, and then executes PLOT such that:

1. All points to be plotted lie in the range of the graph.
2. Gridwork is square.
3. Numeric labels on abscissa and ordinate grid lines are "nice" values.
4. Graph is approximately square.
5. If the points to be plotted have abscissa and/or ordinate values whose magnitude is greater than 10.P.7, the numeric labels to the grid lines are modified by a scale factor, and a heading is printed out informing the user of the size of the scale factor.
6. If the size of the graph is indeterminate in either the Y(vertical) and/or the X(horizontal) direction (ie. a horizontal or vertical line, or a point), an appropriate comment is printed out and the maximum and minimum values of the appropriate axes are adjusted so that the values may be graphed.

RESTRICTIONS: All points (X,Y) which are to be plotted must be obtained and stored in tables before executing SETPLT.

CALLING SEQUENCES: There are two calling sequences available, a regular and an alternate one.

REGULAR CALLING SEQUENCE: User executes only SETPLT (or USTPLT). User does not execute PLOT.

```
MAD         EXECUTE SETPLT.(L,XLOC,YLOC,NUM,BCD,NCHAR,LABEL)
FORTRAN     CALL SETPLT(L,XLOC,YLOC,NUM,BCD,NCHAR,NHABCD...)
UMAP        CALL    USTPLT
            PAR     L
             .      .
             .      .
             .      .
            PAR     LABEL
          or any equivalent UMAP subroutine call
```

ALTERNATE CALLING SEQUENCE: This is for users who want to use "OMIT" to change the graph before it is printed, who want to print more than one copy of the graph, who want to use different plotting characters for different parts of the data, or, in general, who want to take advantage of some of the special features of PLOT (for details on these special features, see the PLOT write-up). When using this alternate calling sequence, user executes SETPLT, and then must execute PLOT3 and PLOT4 (or FPLOT4) himself.

```
MAD         EXECUTE SETPLT.(L,XLOC,YLOC,NUM)
FORTRAN     CALL SETPLT(L,XLOC,YLOC,NUM)
UMAP        (for this alternate calling sequence either the
            name SETPLT or USTPLT may be used.)
            CALL    USTPLT,L,XLOC,YLOC,NUM
            or equivalent subroutine call.
```

ARGUMENTS:

- L — Non-zero if maximum graph length is to be one page. Zero otherwise. (In this case, length .LE. 2 pages)
- XLOC — Location of first value of X or points (X,Y) to be plotted (in table of X values).
- YLOC — Location of first value of Y of points (X,Y) to be plotted (in table of Y values). (These two tables must be stored backwards in storage, as MAD and FORTRAN do. The values of X and Y stored in these must be floating point values.)
- NUM — Number of points to be plotted (either MAD,UMAP, or FORTRAN integer).
- BCD — Left-adjusted BCD(Hollerith) plotting character.
- NCHAR — Number of BCD characters (including blanks) in the LABEL array.
- LABEL — Name of array containing the string of BCD characters to be printed at left edge of output page (label for ordinate). Must be stored backward when using MAD (using vector values statement), or forward when using UMAP (using BCD or BCI block). Must be stored 6 characters to the word (C6).

NHABCD... For FORTRAN users, the string of characters for the ordinate label appears directly in the calling sequence. The N preceeding the H (specifying the Hollerith string) should be the same as the value of NCHAR.

SUBROUTINES REQUIRED: .PRINT, PLOT1, PLOT2, PLOT3, PLOT4, FPLOT4, ELOG, .01301

SAMPLE PROBLEM: This problem is the first example problem at the end of the PLOT write-up, rewritten to use SETPLT. It is suggested that the reader compare them. Both MAD and FORTRAN versions are given.

```
$COMPILE MAD, PUNCH OBJECT                                          PLMAD000
          R
          R PROGRAM TO ILLUSTRATE PLOTTING MULTIPLE POINTS WITH MAD
          R
           DIMENSION X(100), Y(100)
           INTEGER N
FIRST      READ FORMAT ENTR, N
           READ FORMAT DATA, X(1)...X(N)
           READ FORMAT DATA, Y(1)...Y(N)
           PRINT FORMAT TITLE
           EXECUTE SETPLT.(1,X(1),Y(1),N,$*$,32,ORD)
           PRINT FORMAT ABS
           TRANSFER TO FIRST
          R
          R FORMAT STATEMENTS
          R
           VECTOR VALUES ENTR = $I10*$
           VECTOR VALUES DATA =$7F10.4*$
           VECTOR VALUES TITLE =$1H1,S54,15HPLOT OF X VS Y   /1H *$
           VECTOR VALUES ABS = $1HO,S55,14HTHE ABSCISSA X  *$
           VECTOR VALUES ORD =$                    THE ORDINATE Y $
           END OF PROGRAM
$DATA


$MADTRAN, PRINT OBJECT, PUNCH OBJECT                                PLFTR000
C
C     PROGRAM TO ILLUSTRATE PLOTTING MULTIPLE POINTS WITH FORTRAN
C
      DIMENSION X(100),Y(100)
    1 READ INPUT TAPE 7,100, N
      READ INPUT TAPE 7,101,(X(I), I=1,N)
      READ INPUT TAPE 7,101,(Y(I), I=1,N)
      WRITE OUTPUT TAPE 6,102
      CALL SETPLT(1,X(1),Y(1),N,1H*,32,34H                    THE ORDINATE
     X Y )
      WRITE OUTPUT TAPE 6,103
      GO TO 1
C
C     FORMAT STATEMENTS
C
  100 FORMAT (I10)
  101 FORMAT (7F10.9)
  102 FORMAT (1H1,54(1H ),15HPLOT OF X VS Y   /1H )
  103 FORMAT (1HO,55(1H ),14HTHE ABSCISSA X )
      END
$DATA
```

FLOATING POINT SINE AND COSINE

ENTRY POINTS: SIN, COS

PURPOSE: Compute COS(X) and SIN(X) for floating point argument X.

CALLING SEQUENCES:

```
MAD       Y = COS.(X,LOC)
          Y = SIN.(X,LOC)
FORTRAN   Y = COS (X)
          Y = SIN (X)
UMAP      CALL    COS
          PAR     X,T
          NORMAL RETURN - Y IN THE ACCUMULATOR.
          CALL    SIN
          PAR     X,T
          PAR     LOC
          NORMAL RETURN - Y IN THE ACCUMULATOR.
```

ARGUMENTS:

X	Argument in floating point for which the SIN(X) or COS(X) is desired.
Y	The resultant function of the argument X.
T	Optional tag.
LOC	Location for return if error detected. (This argument may be omitted.)

ERROR CONDITION: If X .G. 6.8719477E10, the error procedure is initiated. If LOC is given, control is returned to the caller. Otherwise, the statement "SINCOS ARG TOO LARGE" will be printed and a dump will be given if requested by the user. In the dump, the original argument X will be in -1 for SIN and the argument minus PI/2 will be in -1 for COS.

SUBROUTINES REQUIRED: .EXIT

SKIP TAPE ROUTINE

ENTRY POINTS: SKIP

PURPOSE: Permits programmers writing in compiler languages to skip files and records on scratch tapes with the efficiency of SKPFIL and SKPREC.

CALLING SEQUENCES:

```
MAD       EXECUTE SKIP. (NFILES,NRECDS,NTAPE)
FORTRAN   CALL  SKIP (NFILES, NRECDS, NTAPE)
```

ARGUMENTS:

NFILES	Number of files to be skipped (may be zero).
NRECDS	Number of records to be skipped after skipping the requested number of files (may be zero).
NTAPE	Logical tape on which files and records are to be skipped.

All arguments are of integer mode.

SUBROUTINES REQUIRED: SKPFIL, SKPREC

SIMULTANEOUS LINEAR EQUATIONS

ENTRY POINTS: SLEC, SLEG, SLEM

PURPOSE: SLEC and SLEG solve the system of linear equations A * X = B by factoring A, with interchanges, into a monic lower and an upper triangular matrix. A double back-substitution with compensating interchanges is used to complete the solution. SLEC uses subroutine CROUTP for factoring, and SLEG uses the subroutine GAUSS. SLEM assumes the matrix A has already been factored and performs a double back-substitution with compensating interchanges (these interchanges are based on the vector Y in the calling sequence).

RESTRICTION: These subroutines use BAKSUB to perform the double back-substitution. It is thus necessary that the user check the solution for accuracy, since BAKSUB completely neglects singularity and inconsistent equations. Look at the restriction under subroutine BAKSUB.

CALLING SEQUENCES:

```
FORTRAN   X = SLEC ( N,A,Z,B,Y )
              SLEG ( N,A,Z,B,Y )
              SLEM ( N,A,Z,B,Y )
MAD       X = SLEC.( N,A,Z,B,Y )
              SLEG.( N,A,Z,B,Y )
              SLEM.( N,A,Z,B,Y )
UMAP      CALL   SLEC,N,A,Z,B,Y
                 SLEG,N,A,Z,B,Y
                 SLEM,N,A,Z,B,Y
          RETURN - FLOATING-POINT SWITCH IN AC.
```

ARGUMENTS:

N	Integer dimension of the square matrix A.
A	First element of the matrix. For further information, see the write-up for subroutine GJRDT.
Z	For a successful return, Z will be the solution vector.
B	Right hand side of the system of equations. (Floating point vector)
Y	Interchange record from factorization subroutine. See write-up for subroutine CROUTP.
X	Floating-point switch. 1. Successful computation 0. Overflow during factorization, cannot continue. -1. Overflow during back-substitution, cannot continue.

SUBROUTINES REQUIRED: BAKSUB, CROUTP, GAUSS

GENERAL CONVERSION ROUTINE

ENTRY POINTS: SPREAD, GATHER, FSPRED, FGATHR

PURPOSE: This routine gives the user access within core to the I/O conversion routines (.IOH) normally used in transmitting data to and from the computer. SPREAD (FSPRED) is used to move a region into a list according to a format. GATHER (FGATHR) is used to collect a list into a region according to a format.

CALLING SEQUENCES:

MAD

```
EXECUTE SPREAD. (REGION,FORMAT (THE LIST))
EXECUTE GATHER. (REGION,FORMAT, (THE LIST))
```

Note that (THE LIST) represents the third and following arguments. The length of the LIST is arbitrary and the parentheses are not actually written. The LIST arguments may be single variables, expressions or block parameters. The list behaves as in normal I/O except that values obtained early in the LIST may not be used later in the same LIST.

UMAP

```
CALL SPREAD              CALL GATHER
PAR REGION               PAR REGION
PAR FORMAT,,1            PAR FORMAT,,1
PAR LIST                 PAR LIST
   OR                       OR
BLK LIST-3,,LIST-15      BLK LIST-7,,LIST-10
   ETC.                     ETC.
```

FORTRAN

```
CALL FSPRED (REGION,FORMAT, ( THE LIST))
CALL FGATHR (REGION, FORMAT, (THE LIST))
```

Note that (THE LIST) in FORTRAN may consist of single parameters or expressions only. (Block parameters cannot be compiled in FORTRAN.) The length of the LIST is arbitrary.

ARGUMENTS:

REGION: Base element of area into which LIST will be moved or from which LIST will be extracted. The process is exactly like I/O except that all data remains in core. A unit record is 132 characters for SPREAD. GATHER will consider the length of the list and will transmit the list in groups of 132 (or fewer) characters (6 characters/word). Partial words are filled with blanks.

FORMAT: Base element of FORMAT vector. The FORMAT is assumed to be stored backward in both MAD and FORTRAN. UMAP users may indicate that their FORMAT is stored forward by making the decrement of the FORMAT argument non-zero (see calling sequences above).

(THE LIST): List arguments. See comments above. (6 characters/word)

Remember that a unit record is 132 characters. If the LIST is not satisfied by the time that the FORMAT is exhausted, the next group of words in REGION will be moved and so on until the LIST is satisfied. As usual the LIST determines the end of the processing, rather than the FORMAT.

SUBROUTINES REQUIRED: .IOH

SQUARE ROOT

ENTRY POINTS: SQRT

PURPOSE: Form square root of floating point number X.

CALLING SEQUENCE:

```
MAD       Y = SQRT.(X,LOC)
FORTRAN   Y = SQRT (X)
UMAP      CALL    SQRT
          PAR     X,T
          PAR     LOC
          NORMAL RETURN - Y IN THE ACCUMULATOR.
```

ARGUMENTS:

X	Floating point argument of which square root is desired.
Y	Floating point square root of X.
T	Optional tag.
LOC	Location for return if error detected. (This argument may be omitted.)

ERROR CONDITIONS: If X .L. 0, then the error procedure is initiated. If LOC is given, control returns to the caller. Otherwise, the comment "NEGATIVE SQRT ARG" is printed and a dump is given if requested. X will be in the accumulator in the dump.

SUBROUTINES REQUIRED: .EXIT

HYPERBOLIC TANGENT

ENTRY POINTS: TANH

PURPOSE: Compute the hyperbolic tangent of a floating point number.

CALLING SEQUENCES:

```
MAD       Y = TANH.(X)
FORTRAN   Y = TANH(X)
UMAP      CALL    TANH
          PAR     X,T
          NORMAL RETURN - Y IN THE ACCUMULATOR
```

ARGUMENTS:

X	Floating point argument for which the hyperbolic tangent is to be computed.
T	Optional tag.
Y	Result in floating point.

SINGLE TABLE INTERPOLATION

ENTRY POINTS: TAB

PURPOSE: Given the value of an independent argument X, perform a Kth order interpolation on a table of (X(I),Y(I)) values for the corresponding dependent argument Y.

CALLING SEQUENCES:

```
MAD       Y = TAB.(X,XT,YT,M1,M2,K,N,SW)
FORTRAN   Y = TAB(X,XT,YT,M1,M2,K,N,SW)
UMAP      CALL    TAB
          PAR     X
          PAR     XT
          PAR     YT
          PAR     M1
          PAR     M2
          PAR     K
          PAR     N
          PAR     SW
          NORMAL RETURN - Y IN THE ACCUMULATOR
```

ARGUMENTS:

X	Independent floating point argument X for which the corresponding value Y is desired.
XT	Name of the first entry in the table of floating point independent variables, X(I).
YT	Name of the first entry in the table of floating point dependent variables, Y(I).
M1	Integral number of storage location steps between each entry of the independent variable table. Normally M1 = 1 when the variables are stored in sequential locations.
M2	Integral number of locations between each entry of the dependent variable table. Normally M2 = 1.
K	Integral order of interpolation desired, K .LE. 5.
N	Integral number of entries in the independent variable table (number of pairs (X(I),Y(I))).
SW	Floating point computation switch. SW = 1.0 Normal return, interpolation successful. SW = 2.0 AC or MQ overflow or underflow or divide check - error return.
Y	Floating point dependent variable, the interpolated value for the independent variable X.

SQUARE MATRIX TRANSPOSITION

ENTRY POINTS: TRANS

PURPOSE: Transpose a square matrix.

CALLING SEQUENCES:

```
    MAD       EXECUTE TRANS.(A,N)
    UMAP      CALL    TRANS
              BLK     A,,D
              PAR     N
              NORMAL RETURN
```

ARGUMENTS:

A	The name of the square array to be transposed. For UMAP calls, the matrix must be stored according to MAD rules.
N	The integer degree of the square array.
D	The name of the dimension vector for the array to be transposed. This vector must be set up according to MAD rules.

ARBITRARY MATRIX TRANSPOSITION

ENTRY POINTS: TRANS1

PURPOSE: Transpose an arbitrary matrix.

CALLING SEQUENCES:

```
    MAD       EXECUTE TRANS1.(A,M,N,B)
    UMAP      CALL    TRANS1
              PAR     A
              PAR     M
              PAR     N
              PAR     B
              NORMAL RETURN
```

ARGUMENTS:

A	First element of the array to be transposed. This must be a backwards-stored (MAD type) array.
M	The integer number of rows in the array.
N	The integer number of columns in the array.
B	Temporary storage region to be furnished by the call of length at least (M*N-2)/36 + 1.

CONVERGENCE TESTING IN ITERATION SUBROUTINES

Since many questions have arisen concerning the convergence testing in the interation subroutines, this explanation is being added. UITR1, UITR2, and UITR3 use essentially the same tests for convergence.

1. TEST 1 - Relative Test.
   This test allows a specified percentage of error. It is especially useful when the magnitude of the root is not known. For instance, if an absolute test were used and an EPS of .01 were used, a root near 1,000,000 would probably never pass the test. However, this relative test cannot be used to test for a root of 0 .

   The test is of the form
   ```
   /(X(I)-X(I-1))/X(I)/.LE. EPS1
   ```
   When /X(I)/.G. EPS1, TEST 1 will be used.

2. TEST 2 - Absolute Test.
   This test is usually used for a root near zero, but may be desirable in other cases.

   The test is of the form
   ```
   /X(I)-X(I-1)/.LE. EPS2
   ```
   When /X(I)/ .LE. EPS1, TEST 2 will be used.

3. USE.
   A. Relative Test.
      If a relative test is desired, EPS2 will not be used unless the root becomes less than EPS1.
      Examples EPS1 = .0001
               EPS2 = .00001

   When X(I) .G. .0001, the test is /(X(I)-X(I-1))/X(I)/.LE..0001
   When X(I) .LE..0001, the absolute test is used. /X(I)-X(I-1)/.LE. .00001.

   B. Absolute Test.
      An absolute test will be used whenever /X(I)/ .LE. EPS1.
      Example EPS1 = 2,000,000
              EPS2 = .0001

   When X(I) .LE. 2,000,000 the test will be /X(I)-X(I-1)/.LE. .0001.

SINGLE ITERATION

ENTRY POINTS: UITR1, UITR1A

PURPOSE: Given X = F(X), to find a value of X within a given epsilon of error. If F(X) contains an iteration, this subroutine is not recommended.

CALLING SEQUENCES:

```
    MAD       EXECUTE UITR1.(X,EPS1,EPS2,K)        (SET UP)
          S   X = F.(X)
              I = UITR1A.(X)
    FORTRAN   CALL UITR1(X,EPS1,EPS2,K)            (SET UP)
          J   X = F(X)
              I = UITR1A(X)
    UMAP      CALL UITR1                           (SET UP)
              PAR X
              PAR EPS1
              PAR EPS2
              PAR K
              NORMAL RETURN
          A   COMPUTE X = F(X)
              CALL UITR1A
              PAR X
              NORMAL RETURN - I IN THE ACCUMULATOR.
```

ARGUMENTS:

X — The name of the floating point argument in the equation X = F(X). X contains an initial guess at the time of execution of UITR1.

EPS1, EPS2 — Floating point epsilon values for the error tests. (See UITR, convergence tests for iteration subroutines.) If /F(X(N))/.G. EPS1 then the test /(F(X(N))-X(N))/F(X(N))/.LE. EPS1 is used for convergence. If /F(X(N))/.LE.EPS1, then the test /F(X(N)) - X(N)/ .LE. EPS2 is used. X(N) is the Nth iteration value.

K — Integer maximum number of iterations.

S,J,A — Statement label, statement number, symbolic address, rexpectively, of that portion of the program where the function X = F(X) is calculated. Entry into UITR1A is expected after computation of the function.

I — Computation switch (floating point)

- I = 1.0 Another iteration is required, recompute the function Z = F(X) and return to UITR1A.
- I = 2.0 Normal return, the solution value is in X.
- I = 3.0 Error return, the ratio A is one, where A = (F(X(N))-F(X(N-1)))/(X(N)-X(N-1)).
- I = 4.0 Error return. The specified number of iterations has been exceeded.

NOTE: For a discussion of the convergence tests utilized in UITR1, see the write-up entitled "CONVERGENCE TESTING IN ITERATION SUBROUTINES."

SINGLE ITERATION - INTERVAL HALVING

ENTRY POINTS: UITR2, UITR2A

PURPOSE: Given F(X) = 0 to find a value for X within a given error in a specified interval (A,B).

CALLING SEQUENCES:

MAD
```
EXECUTE UITR2.(A,DELX,B,EPS1,EPS2,K,X)
I = UITR2A.(F)
```

FORTRAN
```
CALL UITR2(A,DELX,B,EPS1,EPS2,K,X)
I = UITR2A(F)
```

UMAP
```
CALL UITR2
PAR A
PAR DELX
PAR B
PAR EPS1
PAR EPS2
PAR K
PAR X
NORMAL RETURN
COMPUTE F
CALL UITR2A
PAR F
NORMAL RETURN - I IN THE ACCUMULATOR
```

ARGUMENTS:

A — Floating point lower limit of the interval (A,B).

B — Floating point upper limit of the interval (A,B).

DELX — The interval (A,B) is stepped across from A, in increments of DELX, until a change of sign occurs in the function F(X). Then this interval is halved a specified number of times until the root is found or the iteration count is exceeded. DELX is floating point.

EPS1, EPS2 — Epsilon values for convergence tests. (See UITR1 write-up.)

K — Integer number of iterations to be allowed.

X — Floating point independent variable. X is the desired root after successful execution of the subroutine.

F — In FORTRAN and MAD, the floating point expression whose value is F(X). In UMAP, F is the location of the value of this function. F must be computed before initial entry into UITR2A.

I — Computation switch - floating point.
I = 1.0 New iteration required. In MAD or FORTRAN, return to UITR2A. In UMAP, recompute the function F and then return to UITR2A.
I = 2.0 The interval (A,B) has been completely scanned and no root was found.
I = 3.0 Number of iterations (K) exceeded without meeting the the test. The current approximate of the root is in X.
I = 4.0 Normal return, computation successful.

NOTE: For a discussion of the convergence tests utilized in UITR2, see the write-up entitled "CONVERGENCE TESTING IN ITERATION SUBROUTINES."

SIMULTANEOUS ITERATION

ENTRY POINTS: UITR3, UITR3A

PURPOSE: Given a set of simultaneous equations of the form

```
X(1) = F(1)(X(1),X(2),...,X(N))
...
...
X(N) = F(N)(X(1),X(2),...,X(N))
```

to find the values of X(I) within a given margin of error. The method and execution of UITR3 corresponds to that of UITR1 except for the number of equations.

CALLING SEQUENCES:

```
MAD       EXECUTE UITR3.(N,K,X,EPS)              (SET UP)
          I = UITR3A.(0)                         (EXECUTION)
FORTRAN   CALL UITR3(N,K,X,EPS)                  (SET UP)
          I = UITR3A(0)                          (EXECUTION)
UMAP      CALL UITR3                             (SET UP )
          PAR N
          PAR K
          PAR X
          PAR EPS
          NORMAL RETURN - I IN THE ACCUMULATOR.
          CALL UITR3A                            (EXECUTION)
          PAR
          NORMAL RETURN - I IN THE ACCUMULATOR.
```

ARGUMENTS:

N — Integer number of equations.

K — Integer number of iterations to be used.

X — The first element of a floating point vector of length at least 4N + 2 in which the X(I) are stored. The vector contains the initial guesses in the first N locations on entry into UITR3A. The answers, of course, appear in the first N locations of the X vector after successful execution of the subroutine.

EPS — The first element of a floating point vector in which EPS1(I) and EPS2(I) are stored to be used for the convergence tests. For any of the given variables X(I), the EPS1(I) and EPS2(I) correspond to the EPS1 and EPS2 in UITR1. The epsilons are stored in order EPS1(1),EPS2(1),EPS1(2),EPS2(2), ..., EPS1(N),EPS2(N).

I — Computation switch - floating point.

- I = 1.0 Successful computation - the solutions are in the first locations of the X vector.
- I = 2.0 Error return - the roots are unobtainable. The N+1 location of the X vector will be 1.0 if the iteration count is exceeded, and it will be 2.0 if the slope of the function is unity. The N+2 location of the X vector contains the number of the equation in which the trouble occurred in floating point.
- I = 3.0 Compute the first function, F(1),and return to UITR3A.
- I = 4.0 Same as 3.0.
- I = 5.0 Compute the second function, F(2), and return to UITR3A.
- I = 6.0 Same as 5.0.
- . . .
- I = 1.0+2.0*N Compute the Nth function and return to UITR3A.
- I = 2.0+2.0*N Same as I = 1.0+2.0*N.

NOTE: For a discussion of the convergence tests utilized in UITR3, see the write-up entitled "CONVERGENCE TESTING IN ITERATION SUBROUTINES."

VARIABLE PRECISION INTEGER ARITHMETIC

ENTRY POINTS: SETUP, CONVRT, ADD, SUB, MPY, DIV, RMNDR, RECNVT, IF

PURPOSE: These subroutines allow the execution of arithmetic on integers whose values range to a maximum of from -10.P.500 to 10.P.500.

USE: The integers each occupy up to a maximum of 50 words of storage. They are to be read in using the format specification MI10, where M is the number of I10 fields needed to contain the integer. The integer, on the input card, must be right-justified in those MI10 fields, and there must be no blanks between any of the digits. The subroutine "SETUP" sets the precision of the arithmetic and the precision of the input. The subroutine "CONVRT" then converts these radix 10.P.10 integers that were read in to radix 2.P.35 integers so arithmetic can be done on them (i.e., it packs them). The subroutines "ADD," "SUB," "MPY,""DIV," and "RMNDR" are available to do arithmetic on the integers. The subroutine "IF" tests an integer in a manner similar to a FORTRAN "IF" statement. The subroutine "RECNVT" converts the integers back to radix 10.P.10 so they can be printed out with I10 formats. See the example program at the end of this write-up.

CALLING SEQUENCES:

```
    MAD       EXECUTE SETUP.(N,M)
              EXECUTE CONVRT.(A)
              EXECUTE RECNVT.(A,Z)
              EXECUTE ADD.(A,B,C)
              EXECUTE SUB.(A,B,C)
              EXECUTE MPY.(A,B,C)
              EXECUTE DIV.(A,B,C)
              EXECUTE RMNDR.(A,B,C)
              K = IF.(A)
    FORTRAN   CALL SETUP(N,M)
              CALL CONVRT(A)
              CALL RECNVT(A,Z)
              CALL ADD(A,B,C)
              CALL SUB(A,B,C)
              CALL MPY(A,B,C)
              CALL DIV(A,B,C)
              CALL RMNDR(A,B,C)
              CALL IF(A,L)
    UMAP      CALL  SETUP,N,M
              CALL  CONVRT,A
              CALL  RECNVT,A,Z
              CALL  ADD,A,B,C
              CALL  SUB,A,B,C
              CALL  MPY,A,B,C
              CALL  DIV,A,B,C
              CALL  RMNDR,A,B,C
              CALL  IF,A
              ----- RETURN WITH K IN THE ACCUMULATOR.
```

VARIABLE PRECISION INTEGER ARITHMETIC (CONTINUED)

ARGUMENTS: All arguments and function returns are integers.

N — One less than the precision of the arithmetic desired. (Precision is the number of storage words each integer can occupy.)

M — One less than the precision of the input values before using CONVRT. Must have 0 .LE. M .LE. N

A,B,C — Vectors at least N+1 locations long reserved by MAD or FORTRAN DIMENSION or UMAP BTS statements (i.e., backwards vectors). The action of the subroutines is:

ADD	C = A + B
SUB	C = A - B
MPY	C = A * B
DIV	C = A / B
RMNDR	C = Remainder of A / B

Z — Vector at least N + 1 +((N+1)/16) locations long reserved by MAD or FORTRAN DIMENSION or UMAP BTS statements.

L,K — Integers returned by 'IF'. L is FORTRAN-type integer, K is full word integer

If A.L.0 then K,L = 1
If A.E.0 then K,L = 2
If A.G.0 then K,L = 3

ERROR COMMENTS: There are three error comments possible. They cover ADD, SUB, or MPY results out of range and attempted division by zero. Control is transferred to .EXIT immediately.

Adapted from Share Distribution No. 1293.

SUBROUTINES REQUIRED: .EXIT

EXAMPLE: This example is written in MAD. It reads in two numbers, A and B, adds them, and if the sum is greater than zero, divides it by 2. Finally, the result is printed out. Note the fact that absolute constants are not to be used as arguments to CONVRT, as CONVRT expects as an argument a vector whose contents it can change.

```
          NORMAL MODE IS INTEGER
          DIMENSION A(20),B(20),X(20),TWO(20)
          READ DATA N,M
          SETUP.(N,M)
          READ FORMAT $7I10*$,A(M)...A(0),B(M)...B(0)
          CONVRT.(A)
          CONVRT.(B)
          TRANSFER TO S(IF.(A))
S(3)      SETUP.(N,))
          TWO=2
          CONVRT.(TWO)
          DIV.(A,TWO,A)
S(1)      CONTINUE
S(2)      RECNVT.(A,X)
          PRINT FORMAT OUT,X(N+1)...X(0)
          VECTOR VALUES OUT = $1HO,20I10*$
          END OF PROGRAM
```

ZEROS OF A COMPLEX POLYNOMIAL

ENTRY POINTS: ZER2, ZER3, ZER4, ZER5, ZER6

PURPOSE: Find the zeros of a polynomial with complex coefficients which is of arbitrary degree, evaluating both real and complex zeros. Roots of multiplicity greater than two are generally unobtainable.

CALLING SEQUENCES:

```
MAD       M = ZER2.(N,A,R)
FORTRAN   M = ZER2 (N,A,R)
UMAP      CALL   ZER2
          PAR    N
          PAR    A
          PAR    R
          NORMAL RETURN - M IN THE ACCUMULATOR.
```

ARGUMENTS:

N — An integer specifying the degree of the polynomial.

A — First element of a floating point vector with the coefficients of the polynomial stored as follows. Assume the polynomial to be of the following form:
P(X) = A(0)X.P.N+A(1)X.P.N-1+A(2)X.P.N-2+...+A(N-1)X+A(N)
Then, first element of A is the real component of A(0).
Second element of A is the imaginary component of A(0).
Third element of A is the real component of A(1).
...
(2N+2)th element of A is the imaginary component of A(N).

R — First element for a floating point array which ZER2 will set to the roots of the polynomial, the first element being the real part of one root, the second element being the imaginary part of the same root, ..., the (2N)th element is the imaginary part of the Nth root.

M — Computation switch - floating point.
M = 1.0 Normal return. The roots are stored in R as described above.
M = 2.0 Error return. Arguments are out of range. AC or MQ overflow.
M = 3.0 Error return. Impossible to locate the roots within the alloted number of iterations (25).
M = 4.0 Error return. First derivative of polynomial at X = X(I) is zero or the coefficient of X.P.N. is zero. (Where X(I) is the Ith value of X in the iteration) i.e., division by zero has occurred.

SPECIAL FEATURES: The special features of ZER2, discussed in this section, are not ordinarily needed. They are available, however, if difficulties develop.

1. For each entry to ZER2, a counter for the number of iterations allowed is set to 25. This initial value may be modified prior to entering ZER2 by specifying the desired number of iterations with the statement:

ZEROS OF A COMPLEX POLYNOMIAL (CONTINUED)

```
MAD       EXECUTE ZER3.(I)
FORTRAN   CALL ZER3(I)
UMAP      CALL ZER3,I
```

where I contains the new iteration count (integer).

2. A succession of three trial initial approximations is allowed for each root in ZER2. If the count is exceeded with the first of these, the second is tried -- if the second also fails, the third is tried. Only if the third also fails is an error return given. The three trial approximations are A+IA, A+IB, A+IC where A=1.0, B=10.0, C=100.0. The quantities A,B,C may be modified by the following statement:

```
MAD       EXECUTE ZER4.(A,B,C)
FORTRAN   CALL ZER4(A,B,C)
UMAP      CALL ZER4,A,B,C
```

where

A+IA = First approximation.
A+IB = Second approximation.
A+IC = Third approximation.

3. In ZER2, if the difference between successive approximations to the real and imaginary parts is less than 2.P.-K times the larger of the characteristics of the approximations - corresponding to a difference in the (27-K) least significant bits of the mantissa - convergence is indicated. ZER2 assumes K=25. This tolerance may be altered by specifying a new K as indicated in the calling sequences in the next section.

4. An essential zero of 10E-9 is used by ZER2 and may be changed to some other value if desired. When either the real or the imaginary component of any approximant becomes less than the essential zero in effect, it is replaced by an actual zero. This essential zero may be any quantity greater than 10E-19 - a smaller essential zero generally leads to accumulator underflow. This mantissa test and the essential zero may be modified by the following statement:

```
MAD       EXECUTE ZER5.(K,S)
FORTRAN   CALL ZER5(K,S)
UMAP      CALL ZER5,K,S
```

where

K = Integer number for the mantissa tolerance test and must be greater than zero and less than 27.
S = Floating point number for essential zero.

5. It is possible to trace the successive approximations to the roots and note their convergence. The approximations come in sets of four parts. The first part is the real component of the first or previous approximation. The second part is the imaginary member of the same approximation. The third part is the real component of the current approximation following the first or previous approximation. The fourth part is the imaginary member of the same approximation. These approximations are obtained by the following statement:

ZEROS OF A COMPLEX POLYNOMIAL (CONTINUED)

```
MAD       EXECUTE ZER6.(C)
FORTRAN   CALL    ZER6(C)
UMAP      CALL    ZER6,C
```

where C denotes where to store the set of approximants - they are stored backwards in core starting at C. The return M from ZER2 will be 1.0, 2.0, 3,0, or 4.0 for normal and error returns (as previously described) or M will be 5.0 to indicate a return with a new set of approximants stored in C. As each new set is obtained from ZER6, the previous set stored at C is destroyed. To continue the iteration for the roots, control must be transferred back to the calling sequence for ZER2. To stop taking the approximates, the following statement should be given:

```
MAD       EXECUTE  ZER6(0)
FORTRAN   CALL  ZER6(0)
UMAP      CALL  ZER6,=0
```

The calling sequence of the special features affects only the items specified - no computations are performed until the subroutine is entered in the normal manner (i.e., via a call for ZER2). Once the routine is modified for one or more of the special features, it remains in that state until restored by the user. A return to the next instruction occurs after each of these special feature entries.

STORE CONSTANT

ENTRY POINTS: ZERO, SPRAY

PURPOSE: ZERO Stores zero.
SPRAY Stores arbitrary constant.

CALLING SEQUENCES:

```
MAD       EXECUTE ZERO.(L1,L2,.....,LN)
UMAP      CALL    ZERO
          L1
          L2
          .
          .
          .
          LN
```

ARGUMENTS:

The LI are standard argument list elements of the form

```
MAD     A...B  or  A
UMAP    BLK  A,,B  or  PAR A .
```

SPRAY is called exactly as ZERO except that the first argument is a single constant (in MAD) or the location of a single constant (in UMAP) which is to be stored instead of zero.

I/O SUBROUTINES

I. INTRODUCTION.

This write-up describes the usage and the structure, but not the interconnection of the I/O routines that provide conversion via a format. For a description of the interconnection, see Appendix VI of this section. This write-up assumes that the reader is acquainted with MAD and/or UMAP.

II. I/O SUBROUTINE CALLING SEQUENCES.

In this section, the means of calling on the I/O subroutines from MAD and UMAP are given. In the case of MAD, only the statement type is given. Details may be found in the MAD MANUAL. In the case of UMAP, only one of the numerous ways to write the subroutine call is given. For further details, see the UMAP write-up in this manual. Note that in UMAP, the pseudo-operations IOP, FMT, TAPENR and ENDIO all translate as STR.

A. GENERAL STRUCTURE OF THE MAD/UMAP CALLING SEQUENCES.

1. The first word is a TSX via index register 4 to the I/O routine. E.G., TSX .PRINT,4.
2. If a tape number must be specified, the address of the next STR specifies the tape number as a full word integer. E.G., STR = 4 .
3. The next STR can be of two types. If the decrement is greater than 1, it is assumed to be type II. Otherwise, it is assumed to be type I.

   TYPE I Address is the location of format. Decrement is the direction in which the format is stored in core. (0 means forward, 1 means backward.)

   TYPE II Address is the location of symbol table. Tag is the direction in which the format is stored in core. (0 means forward, 1 means backward.) Decrement is the location of format.

   Typical UMAP usage is FMT FORM. MAD typically generates what could be written in assembly code as

   ```
   STR  STLOC,1,FORM.
   ```

   Type II usage is necessary only if format variables are used.
4. The following word(s) contain the list. (See the section on list structures for a discription.)
5. The last word is a STR with zero address and decrement. This is the signal for termination of the calling sequence.

In MAD, the calling sequence is set up in this form by the translator. In UMAP, the user is responsible for putting the calling sequence in the correct form.

B. SPECIFIC CALLING SEQUENCES.

1. Reading cards from system input tape.

```
MAD -       'READ FORMAT'
UMAP -      CALLIO  .READ
            FMT     FORMAT
               (LIST)
            ENDIO
```

I/O SUBROUTINES (CONTINUED)

This subroutine causes BCD information to be read from the system input tape and converted to binary, according to the format specification. Since it is written on the input tape in card-image form, the format specification may not describe more than 80 columns. If an end-of-file is found (i.e., no more data cards), the job will be terminated (unless SETEOF has been executed. See the writeup for SETEOF in this section of the manual).

2. Printing lines on system output tape.

```
MAD -       'PRINT FORMAT'
UMAP -      CALLIO .PRINT
            FMT    FORMAT
               (LIST)
            ENDIO
```

This subroutine causes the binary information indicated by the list to be converted to BCD according to the format and written on the system output tape as lines to be printed. Since the information is written in line-image form on the tape, the format specification may not describe more than 132 columns per line.

3. Looking at cards from system input tape.

```
MAD -       'LOOK AT FORMAT'
UMAP -      CALLIO .LOOK
            FMT    FORMAT
               (LIST)
            ENDIO
```

This is the same as reading cards, but without going past the card. Hence, the next time a 'READING CARDS' or 'LOOKING AT CARDS' I/O call is processed, the same card will be again transmitted. The format given can only specify one card. If more than one is specified (via one or more slashes in the format or format termination with list unsatisfied), each instruction to get a new card causes the same card to be rescanned.

4. Punching cards on system punch tape.

```
MAD -       'PUNCH FORMAT'
UMAP -      CALLIO .PUNCH
            FMT    FORMAT
               (LIST)
            ENDIO
```

This subroutine causes the binary information indicated by the list to be converted to BCD according to the format and written on the system punch tape as cards to be punched. Since information is written in card-image form on the tape, the format specification may not describe more than 80 columns.

I/O SUBROUTINES (CONTINUED)

5. Reading tape (input from arbitrary tape).

```
MAD -       'READ BCD TAPE N'
UMAP        CALLIO .TAPRD
            TAPENR  =N
            FMT     FORMAT
               (LIST)
            ENDIO
```

This subroutine causes BCD information to be read from the specified tape unit and converted to binary form according to the format. In the MAMOS system, the only tape units that can be specified for this subroutine are 2,3,4,7,9,10 and 11. Tape 7 is the input tape, and the others are available as scratch tapes. If tape 7 is specified, the format may not describe more than 80 columns. For other tapes the format may not describe more than 132 columns. If an end-of-file is encountered during reading tape, the job will be terminated unless the subroutine SETEOF has been executed. See the SETEOF write-up in this manual.

6. Writing tape (output on arbitrary tape).

```
MAD -       'WRITE BCD TAPE N'
UMAP        CALLIO .TAPWR
            TAPENR  =N
            FMT     FORMAT
               (LIST)
            ENDIO
```

This subroutine causes the binary information indicated by the list to be converted to BCD form according to the format and written on the specified tape unit. In the MAMOS system, the only tape units that can be specified for the subroutine are 2,3,4,5,6,9,10,11. Tape 6 is the print output tape. Information written on it will be printed on the off-line printer, and, hence, the format must not specify more than 132 columns. Tape 5 is the punch output tape. Information written on it will be punched on cards and, hence, the format must not specify more than 80 columns. For the other tapes, the format must not specify more than 132 columns. If the end of tape is encountered during writing, the tape is rewound, a comment is printed to the operator, and the computer stops to allow the tape to be replaced. The writing of information continues on the new tape (without loss of information). This procedure can be modified by using the subroutine SETETT. See the SETETT write-up in this manual.

7. Printing on the on-line printer.

```
MAD -       'PRINT ON LINE FORMAT'
UMAP -      CALLIO .COMNT
            FMT     FORMAT
               (LIST)
            ENDIO
```

This subroutine causes the binary information indicated by the list to be converted to BCD according to the format and printed on the on-line printer. It is to be used only for comments to

I/O SUBROUTINES (CONTINUED)

the operator, not for output. The format specification should not contain a carriage control and may not describe more than 72 columns.

III. STRUCTURE OF A LIST UNDER MAD/UMAP

The list designates locations whose contents are to be converted and transmitted. In MAD, the lists are automatically produced by the translator. In UMAP, the user is responsible for putting them in the correct form. List words are of two types - single variables and blocks.

1. Single variable - the list word is of the form

```
    IOP    A
or  IOP    A,T
```

'T' may refer to any of the three index registers, but the user should remember that the 'TSX' to the subroutines has changed the value of index register 4.

2. Block - this is used to designate an entire region (say, from A to and including B) of consecutive locations. The form of the list word is

```
    IOP    A,,B
```

The entire region is transmitted, starting with A and ending with B, both when A is less than B and when B is less than A.

A typical list might be

```
    IOP    ALPHA
    IOP    V,,V+10
    IOP    R-5,,R+7
    IOP    A,,A-100
    IOP    DENOM
```

and a typical output calling sequence for this might then be

```
    CALLIO .PRINT
    FMT    FORM1
    IOP    ALPHA
    IOP    V,,V+10
    IOP    R-5,,R+7
    IOP    A,,A-100
    IOP    DENOM
    ENDIO
```

The reason for the operation 'IOP', which is translated as the operation "STR', lies in the use of the trapping mode. When an STR is executed, the contents of the instruction location counter, which is pointing to the instruction following the 'STR', is put into location 0, and the computer transfers to location 2. Previous to this, the I/O routine has inserted into location 2 a transfer to its internal subroutine where it gets a new list element. The subroutine doing the conversion scans the format (see section IV - format scan action) until it needs a value of some entry on the list. It then does a transfer indirect to location 0, which effectivly causes a transfer to the location one past the last entry on the list. In

I/O SUBROUTINES (CONTINUED)

the above example, this is always an 'IOP' ('STR') operation and it is trapped back to the subroutine, at the same time giving its location to the subroutine (contents of location 0), so the subroutine can obtain the address of the next variable (or region) on the list. This process is continued until the 'ENDIO' ('STR' with zero address and decrement) is found which causes an entry to that part of the subroutine which terminates input/output and then returns to the instruction immediately following the 'ENDIO', thus going back to the user's program.
It follows that the user may, if he desires, make the list much more complicated. Consider the following list:

```
        IOP     ALPHA
        IOP     V,0,V+5
        AXT     1,4
    Q   IOP     W+1,4
        TXI     *+1,4,2
        TXL     Q,4,60
        IOP     C
```

As before, the value of ALPHA and of V through V + 5 will be converted and transmitted. The subroutine will return for another value, but will instead set index 4 equal to one, then execute the IOP (at location Q) and get trapped so as to furnish the subroutine with the effective address of W. The subroutine returns for another value and we increase index 4 by 2 and test it against the upper bound of 60. If not yet finished with the loop, we go back to Q and repeat the loop until the contents of index register 4 exceed 60. When this occurs, the next variable furnished to the subroutine is C. There is no limit to the amount of computation one can perform between the 'IOP's, but an ENDIO must eventually occur.

IV. FORMAT SCAN ACTION.

When a user calls a specific I/O routine (such as .PRINT), this routine sets up certain parameters (such as location of format, maximum number of columns allowed, etc.) in a communication region and then calls on the subroutine .IOH which does the format scan.

The scan begins at the first character of the format and in normal context. It scans character by character from the beginning of the format toward the end, setting switches and parameters, until a break character is found or a change of context is signalled.

When a break character is found, processing is begun of the format term whose end is indicated by the break character. If the format term is a data-transmission format term, reference to the list is made. If input, the list is referenced to get the value to be converted, and then the conversion is done. If output, the conversion is done, and then the list is referenced to get the location where the converted value is to be put. In all cases, if there are no more list words when the list is referenced (list exhausted), the format scan stops and the I/O statement is terminated, just as if the format terminator had been hit. If the format term is a non-data-transmission format term, it is processed immediately. When processing of the format term is complete, switches and parameters are reset, and the format scan continues with the next character.

I/O SUBROUTINES (CONTINUED)

When a change of context is signalled, processing of this new context begins immediately. When the end of the format is sensed, if it is an output format, an output record is produced (i.e., line is printed, card is punched, etc.). Then the list is inspected to see if it is exhausted (all items processed). If so, then the I/O call is terminated. If not, .IOH begins rescanning the format, in the same direction as before. If no parentheses were used in the format, the scan restarts from the beginning of the format. If parentheses were used, the format scan restarts at the right-most zero-nesting-level left parenthesis, using its multiplicity, if any. (Zero-nesting-level means that it is inside no other parentheses).

V. MEANINGS OF EACH CHARACTER IN FORMATS.

Since the meaning of a character in a format depends on the context it is in, a separate listing of character meanings is provided for each context. For each of the contexts other than normal one, a statement of its purpose and the way entry to and exit from this context is specified are given. It should be noted that context changes are only to or from normal context.

A. NORMAL CONTEXT.

The characters are listed here in ascending order according to their octal representation. Note that in all cases where multiplicity applies, omitting the multiplicity is equivalent to giving a multiplicity of 1.

--- 0 (00 OCTAL) DIGITS
1 (01 OCTAL)
2 (02 OCTAL)
3 (03 OCTAL)
4 (04 OCTAL)
5 (05 OCTAL)
6 (06 OCTAL)
7 (07 OCTAL)
8 (10 OCTAL)
9 (11 OCTAL)

Whenever a digit is encountered it causes the accumulation of a number to begin or continue. These numbers, depending on their position, are interpreted as field widths, multiplicities, etc.

--- (12 OCTAL)
= (13 OCTAL)

These are illegal characters.

--- ' (14 OCTAL) PRIME

The prime signifies a change from normal to format variable context, and from format variable context to normal context.

--- (15 OCTAL)
(16 OCTAL)
(17 OCTAL)

These are illegal characters.

--- + (20 OCTAL) PLUS SIGN

A plus sign is ignored.

--- A (21 OCTAL)

BCD control character - 'A' and 'C' are interchangeable. See description under 'C'. (There are two characters for the same thing because 'A' has been traditionally used in FORTRAN formats and 'C' in MAD/UMAP formats.)

I/O SUBROUTINES (CONTINUED)

--- B (22 OCTAL)

Variable base modifier. Normally, conversion is done from binary in the machine to a decimal (base 10) external form, and from a decimal external form to binary in the machine. The user may specify other external form bases for integers by including the modifier N B where N is the conversion base wanted. N must be greater than 1 and less than 20. For those bases greater than 10, the additional characters needed are taken from the beginning of the alphabet. For base=19 (the largest possible) the characters used are (in ascending order)- 0 1 2 3 4 5 6 7 8 9 A B C D E F G H I. For example, 5B2I10 will cause 2 integers to be read or printed in base 5.

NOTE 1. 8BI-- in a format is not the same as K--, since for integers the left-most bit is considered a sign, whereas for octal numbers it is considered as part of the left-most digit. Also, octal conversions in which the converted number exceeds the specified field width are truncated, but integer conversions that exceed the field width are errors.

NOTE 2. The numbers found in the formats themselves are always taken to be base 10.

--- C (23 OCTAL)

BCD control character. The control characters 'C' and 'A' are used to read in and print out BCD information (characters). In the 7090/7094 a character is represented by 6 bits, so there are 6 characters to a machine word. The BCD specification assumes that whatever is named on the list to be transmitted exists in the machine as characters. The basic format term is of the form N C W where N is the multiplicity and W the field width. To see how transmission occurs, imagine the left end of the word in storage lined up with the left end of the specified field. For input, as many characters as the field width specifies are moved from the field directly into the storage location. If fewer than six, they fill in the left end of the word and blanks are used to fill out the rest of the word. If more than six, the first six fill up the word and the rest are lost. For output, as many characters as the field width specifies are moved from the storage word directly into the field. If fewer than six, the left-most characters are used. If more than six, the six characters in the word are put in the left-end of the field and blanks are used to fill out the rest of the field.

For example, if a card contains the characters ABCDEFGHIJK in columns 1 through 11, and it is read in according to the specification 2C3*, the two 6 character words that are read into the computer are:

ABC (with three trailing
DEF blanks on each)

while the specification C6* would cause a single word to be read:

ABCDEF

and C7,C3* would cause the words

ABCDEF and HIJ (with 3 trailing blanks)

to be read.

I/O SUBROUTINES (CONTINUED)

FORM OF THE INPUT FIELD - There is no form. All characters are allowable. A blank is like any other character and is not ignored. Fields come in left-justified with trailing blanks.
FORM OF THE OUTPUT FIELD - There is no form. Fields go out left-justified with trailing blanks.
APPLICABLE MODIFIERS - R,W,Z

--- D (24 OCTAL)

Double-precision modifier. If the modifying character 'D' appears in an E,F, or G format term, it indicates that this format term refers to a double-precision number, and conversion will be carried out in double precision. Standard 7094 double-precision form is assumed in that the number is contained in two machine words, each is a complete floating point number with exponent and fraction, and the exponent of the low order half is 27 smaller than the exponent of the high order half. Both halves must be named on the list, the high order half first. The precision of single precision floating point numbers is 8 significant digits. For double precision numbers it is about 16. (Significant digits means considering all the digits, not just those after the decimal point. A single precision number of magnitude 10^8 when converted by an F-type specification would have 8 digits in front of the decimal point, hence any digits that appeared after the decimal point would not be significant.) The range of double precision numbers is the same as the range of single precision numbers, about 10^{38} to 10^{-38}.

--- E (25 OCTAL)

E-type floating point control character. E-type conversion assumes that the number named on the list is a floating point number in storage. The form of the format term is N E W.D where N is the multiplicity, W is the field width, and D is the number of digits after the decimal point.
FORM OF THE INPUT FIELD - An E-type input number as it appears on an input card must have the form +XXX.XXXXXE+YY . The sign and digits following the E represent the exponent of 10 by which the number in front of the E is to be multiplied. I.e., this number means

$$+XXX.XXXXX \text{ TIMES } 10^{\pm YY}$$

The sign of the fractional part may be omitted if it is positive. If it is negative, the minus sign must be included. Any number of digits may be used in the fractional part, but only 8 digits of accuracy are retained. If the decimal point is present, the 'D' in the format term is ignored. If the decimal point is not present, the 'D' specifies that the right-most 'D' digits of the fractional part come after the decimal point. Hence the punched number +9032E3 described by the specification E10.4 would be understood to be the number +.9032E3. The exponent must be included in the field width. If the sign of the exponent is included, the E may be omitted. If the E is present, the + sign may be omitted for positive exponents. If the exponent is to be negative, the minus sign

I/O SUBROUTINES (CONTINUED)

must be included. The exponent must be within the limits of the 7090/7094 between about +38 and about -38. Leading zeroes on the exponent may be omitted. Blanks are ignored throughout the whole field. An all blank field is read in as -0. Numbers in E input fields may be either the E-type described above or the F-type to be described under the control character F.

FORM OF THE OUTPUT FIELD - Numbers printed or punched in E fields have the form (if 5 decimal digits are requested, for example) ±.XXXXXE±XX although + signs are not produced. The number is rounded, not truncated, to the number of digits wanted after the decimal point. The sign, decimal point, 'E' and exponent must be included when figuring the field width, so W must be greater than or equal to D+5 (if the numbers will always be positive) or D+6 (if the numbers could be negative). If the number when converted requires more columns than the field width allows, an error comment is printed and the job is terminated. (To modify this procedure, see Appendix I to this write-up and see the write-up on IOHSIZ in this manual.) If the converted number requires fewer columns than the field width specifies, the number is right-justified in the field. In fact, some spacing can be achieved by giving large field sizes, since blanks automatically occur to the left of a number pushed to the right end of an oversized field.

APPLICABLE MODIFIERS - D,L,M,P,V,W.

--- F (26 OCTAL)

F-type floating point control character. F-type conversion assumes that the number named on the list is a floating point number in storage. The form of the format term is N F W.D or N F W where N is the multiplicity, W is the field width, and D is the number of digits after the decimal point. The shorter form is assumed to be equivalent to the longer form where D is 0. I.e., F6 is the same as F6.0.

FORM OF THE INPUT FIELD - An F-type number as it appears on an input card must have the form ±XXX.XXXXX. The + sign may be omitted if the number is to be positive, but if the number is to be negative, the minus sign must be there. Any number of digits may be used, but only 8 digits of accuracy are retained (due to the limits of the 7090). Blanks are ignored. An all blank field is read in as -0. If the decimal point is present, the 'D' in the format term is ignored. If the decimal point is not present, the 'D' specifies that the right-most 'D' digits come after the decimal point. Hence the punched number +9032 described by the format term F10.2 would be understood to be the number +90.32. Numbers in F input fields must either be the F-type described above or the E-type described previously.

FORM OF THE OUTPUT FIELD - Numbers printed or punched in F fields have the form (if 5 decimal digits are requested, for example) ±XXX.XXXXX although + signs are not produced. The number is rounded, not truncated, to the number of digits wanted after the decimal point. If no digits are produced after the decimal point, the point itself is not produced. If the number when converted

I/O SUBROUTINES (CONTINUED)

requires more columns than the field width allows, an error comment is printed and the job is terminated. (To modify this procedure, see Appendix I to this write-up and see write-up on IOHSIZ in this manual.) If the converted number requires fewer columns than the field width specifies, the number is right-justified in the field. In fact, some spacing can be achieved by giving large field sizes, since blanks automatically occur to the left of a number pushed to the right end of an oversized field.
APPLICABLE MODIFIERS - D,L,M,P,V,W.

--- G (27 OCTAL)

Significant digits control character. This is exactly the same as the F field, except that on output, instead of having the decimal point fixed and the significant digits vary around it, the number of significant digits printed out is fixed and the decimal point floats. The form of the format term is N G W.D where N is the multiplicity, W is the field width, and D is the number of significant digits wanted. This usually includes the decimal point and, possibly, a minus sign, so only D-1 or D-2 digits are actually produced. If the decimal point is not present, it has vanished off the right end of the field. The D characters are right-justified in the W columns of the field width. The number is first converted entirely, and then the left-most D characters are put out as output. Hence, the number put out is a truncation, not a rounding, of the complete number. D must be less than or equal to W. If D is greater, an error occurs (see Appendix I of this write-up and write-up IOHSIZ in this manual for modification of this procedure). For example, the numbers 12345., 1234.5, 12.345, .12345, and .00123, printed according to 5G6.4, would give 1234 1234 12.3 .123 and .001.
FORM OF THE INPUT FIELD - Same as for F fields.
FORM OF THE OUTPUT FIELD - Same as for F fields.
APPLICABLE MODIFIERS - Same as for F fields.

--- H (30 OCTAL)

Hollerith control character. An H causes a change of context from normal context to Hollerith context.

--- I (31 OCTAL)

Integer control character. This conversion assumes that the numbers named on the list exist in storage as full-word integers. The form of the basic format term is N I W where N is the multiplicity and W is the field width.
FORM OF THE INPUT FIELD - The number in an I-type input field must be of the form (for example) ±XXXXX. If the number is to be positive, the + sign need not be punched. If the number is to be negative the minus sign must be punched. The only legal characters in the body of the integer are those digits and letters which are less than the base. Unless otherwise specified, the base is 10 (decimal) and so the only legal characters are 0 through 9. Blanks are ignored. An all blank field comes in as -0.

I/O SUBROUTINES (CONTINUED)

FORM OF THE OUTPUT FIELD - The number produced by I-type conversion is of the form (for example) +XXXXX except that + signs are not produced. If the number when converted requires more columns than the field width allows, an error comment is printed and the job is terminated. (See Appendix I of the write-up and the write-up on IOHSIZ in this manual for modification of this procedure.) If the converted number requires fewer columns than the field width specifies, the number is right-justified. Since blanks occur automatically to the left of a right-justified number, some spacing can be achieved by giving an oversized field.
APPLICABLE MODIFIERS - B,L,M,V,W.

--- +0 (32 OCTAL)
This is an illegal character.

--- . (33 OCTAL) PERIOD
Punctuation. The period is used in E, F and G specifications. It tells the format scanner that the number accumulated so far is to be considered the field width, and that a new number is to begin accumulating.

---) (34 OCTAL) RIGHT PARENTHESIS
Multiple grouping. See left parenthesis (OCTAL 74) for description of using parentheses.

--- (35 OCTAL)
(36 OCTAL)
(37 OCTAL)
These are illegal characters.

--- - (40 OCTAL) MINUS SIGN
This causes the sign of the number being accumulated by the format scanner to be reversed.

--- J (41 OCTAL)
This is an illegal character.

--- K (42 OCTAL)
Octal control character. This mode of conversion makes no assumption about the form of the number in storage. It just reads in or prints out the number as octal. The form of the basic format term is N K W where N is the multiplicity and W is the field width.
FORM OF THE INPUT FIELD - A K-type input field is of the form ±XXXXX . The sign is optional. There must be no more than 12 digits in the number, and each digit must be one of 01234567. Blanks are ignored. An all blank field comes in as +0. The transmitted numbers are right-justified in the machine word with leading zeroes.
FORM OF THE OUTPUT FIELD - A K-type output field is of the form XXXXX. No sign is produced. If W (field width) is less than 12, the rightmost W digits of the word are put out. If W is greater than 12, the full word is put out in the field, right-adjusted. Since blanks occur to the left of a right-adjusted number, this provides a means of spacing the number.
APPLICABLE MODIFIERS - L,W.

I/O SUBROUTINES (CONTINUED)

--- L (43 OCTAL)

Left-adjusted modifier. This modifier changes things as follows:

K,O FIELDS

Input

Normally the number is right-justified with leading zeroes in the machine word. With the L modifier, it is left-justified with trailing zeroes.

Output

Normally the number is placed at the right end of an oversized field. With the L modifier it is placed at the left end, with blanks filling the unused portions to the right.

E,F,G,I FIELDS

Output

Normally the number is placed at the right end of an oversized field. With the L modifier it is placed at the left end, with blanks filling the unused portions to the right.

--- M (44 OCTAL)

Floating dollar sign modifier. When this modifier is used in E, F, G or I output format terms, a dollar sign is inserted into the field immediately to the left of the first digit. If the number is negative (so there is a minus sign), the dollar sign goes immediately to the left of the minus sign.

--- N (45 OCTAL)

"Don't" modifier. If the N modifier occures before a slash in an output format, it means don't blank out the line and don't reset the line pointer to column 1 after printing. If the N modifier occurs before the format terminator in an output format, it means don't print the line or reset it when exiting (if the list is exhausted). These two applications of 'N' along with the subroutine STQUO (see section VII) will allow building up a print line column by column, adding each number with a separate I/O statement, but not printing the line until it is complete. This avoids using the + carriage control and wasting printer time.

--- O (46 OCTAL)

Octal control character. The control characters 'K' and 'O' are interchangeable. See the description under 'K'. (The reason for the two control characters meaning the same thing is that FORTRAN formats have traditionally used 'O' and MAD/UMAP formats 'K'.)

--- P (47 OCTAL)

Scale factor modifier. This feature is allowed for E and F fields. A scale factor may be applied to an F number according to the formula EXTERNAL NUMBER = INTERNAL NUMBER X 10^P (where the scaling is accomplished before the conversion is done). The scale factor followed by the letter P is prefixed to the basic field specification as in the example 2P2F7.3,F7.3* . Thus, three numbers which would print .522 -1.567 93.671 according to the specification 3F7.3* would print instead .005 -.016 93.671 if the specification -2P2F7.3, F7.3* were used. It must be noted that this scale factor actually changes the values of the numbers to which it applies. It affects

I/O SUBROUTINES (CONTINUED)

only those numbers to which it is directly applied, however. For E fields, the scale factor causes the number itself to be modified, but the exponent is correspondingly modified so the true value of the number remains unchanged. Thus, the number .9321E-3 would print as 93.2100E 05 according to the specification 2PE16.4*. Unlike an F number, the value is the same in either case.

--- Q (50 OCTAL)

This is an illegal character.

--- R (51 OCTAL)

Right-justified modifier. Normally, characters read and printed with A and C control characters are left-justified, both in the machine word and in the field. If the R modifier is used, on input the characters are right justified in the machine word (with leading blanks), and on output the characters are right justified in the field. The description of how it works in this case is the same as for L, except that 'LEFT' should be replaced by 'RIGHT' wherever it occurs.

--- $\bar{0}$ (52 OCTAL)
     $ (53 OCTAL) DOLLAR SIGN

These are illegal characters.

--- * (54 OCTAL) ASTERISK

Format terminator. The asterisk terminates the format scan for MAD/UMAP calls.

APPLICABLE MODIFIERS - N.

--- (55 OCTAL)
     (56 OCTAL)
     (57 OCTAL)

These are illegal characters.

--- BLANK(60 OCTAL)

Blanks are ignored.

--- / (61 OCTAL) SLASH

New line or card control character. If input, a / causes a new input record to be read in and the line pointer is reset to column 1. If output, a / causes an output record to be sent out, and then the line-image is blanked out and the line pointer reset to column 1.

APPLICABLE MODIFIERS - N.

--- S (62 OCTAL)

Skip control character. The form of the format term is N S W where N is the multiplicity and W is the number of columns to skip. The action of this format term is to add W to the line-pointer. W may be either positive or negative. E.g., to produce "BA" in an output line-image by putting in 'A' first and then backspacing and putting in 'B', the format must contain 1HA,S-2,1HB.

WARNING - When a print line or similar output record is produced, it is assumed that the last column produced is immediately to the left of the line pointer and only enough words to contain this column are put out. (Each word contains 6 columns.) If negative skips or backwards transfers have occurred, some of the line may fail to get printed. To avoid any difficulties, the format term T132 can be inserted as the last one before the line is printed.

I/O SUBROUTINES (CONTINUED)

--- T (63 OCTAL)
Transfer (or tabulator) control character. The form of the format term is T N where N is a column number. This format term causes the line-pointer to be reset to N. The warning given for the 'S' character also applies here.

--- U (64 OCTAL)
This is an illegal character.

--- V (65 OCTAL)
Commas every three digits modifier. When this modifier is included in an E, F, G, or I output format term, it causes commas to be placed every three digits to the left of the decimal point (or to the left of the right end of the number, if there is no decimal point). These commas must be allowed for in the specification of the field width.

--- W (66 OCTAL)
Blank if zero modifier. If the modifier occurs in an A,C,E,F,G,I,K, or O output format term, any number that is all zero will not be printed out. Instead, blanks will be put in its place. Note that this does not mean that zeros cannot be printed out. A number may very possibly be not all zero and yet, when converted, round to zero.

--- X (67 OCTAL)
Space control character. The form of the format term is N X where N is the number of columns to be spaced. Its action is the same as N S1.

--- Y (70 OCTAL)
This is an illegal character.

--- Z (71 OCTAL)
Leading or trailing zeros modifier. BCD information that is read in normally ends up with trailing (or leading, if an R modifier was in the format term) blanks. If a Z modifier is included in the format term, this will cause trailing (or leading) zeros, rather than blanks. E.G., RZC1 reading in the letter 'A' from a card will give a word with 000000000021 (OCTAL) in it.

--- ‡ (72 OCTAL) RECORD MARK
This is an illegal character.

--- , (73 OCTAL) COMMA
Punctuation. The comma separates format terms.

--- ((74 OCTAL) LEFT PARENTHESIS
Punctuation. A group of format terms may be repeated by enclosing the group in parentheses and preceding the left parenthesis by the multiplicity. Thus 3E10.3, 2(I2,3F10.1),2C5* is equivalent to E10.3,E10.3,E10.3,I2,F10.1,F10.1,F10.1,I2,F10.1,F10.1,F10.1,C5,C5*. Nested parentheses are allowed. There is no limit to the nesting depth. However, information about parentheses is kept in a push-down list in erasable, with each nesting level causing a two word entry. Hence, the deeper the nesting depth, the more erasable storage is used. If the multiplicity in front of a left parenthesis is zero, this means do what is inside zero times which means do not do it. This causes a switch from normal context to format-off context.

I/O SUBROUTINES (CONTINUED)

--- (75 OCTAL)
(76 OCTAL)
(77 OCTAL)
These are illegal characters.

B. HOLLERITH CONTEXT.
Hollerith context is entered when an H is encountered in normal context. The purpose of the Hollerith context is to provide characters in the format which can be put in the print-line, or, alternatively, replaced by characters from the card image. This is used for titles, labels and other constant information. If there was a count in front of the H that signalled the change to Hollerith context, it is a counted Hollerith context. Otherwise it is uncounted.

COUNTED - The first character in the Hollerith context is the one immediately following the H. The last character in the Hollerith context is the Nth character following the H (where N is the count).

OUTPUT - All the characters in the Hollerith context are moved into the line-image, where they form a Hollerith field. The first goes into the line-image at the column the line-pointer is pointing to, the line-pointer is advanced one column, the second goes into the line image at the column the line-pointer is pointing to, and so on. Thus 8HTRIANGLE will cause the word "TRIANGLE" to be put into the output line.

INPUT - All the characters in the Hollerith context are replaced by the characters in the line-image. The first is replaced by the character in the column the line-pointer is pointing to, the line-pointer is advanced one column, the second is replaced by the character in the column the line-pointer is pointing to, and so on. Thus, a card punched as follows:

1 DATA SET NO. 3-A JULY 19, 1963

might be read in with a format specification

72H (72 blank spaces) *

Later, this specification could be used to print the same information as a heading for the results.

WARNING: The specifications S72* and 72(1H)*, while indicating 72 blank spaces, do not allow the reading in of an entire card, as indicated above, since they do not provide a region of 72 characters in length in the format into which the information on the card may be read.

UNCOUNTED - The first character following the H is taken as a break character, and all characters between it and the next occurance of this same break character are considered to be in the Hollerith context. Output and input are exactly the same as given above. The two examples given above, when written to use uncounted Hollerith contexts, are H*TRIANGLE* and

H= (72 blank spaces) =*

I/O SUBROUTINES (CONTINUED)

C. FORMAT VARIABLE CONTEXT.
Use of the format variable context allows substituting the value of a variable in the program making the I/O call into a format anywhere a number would otherwise be placed. This substitution takes place at the time the format variable context is encountered during the scan of the format. A prime is the signal both for entry to and exit from format variable context, i.e., primes delimit format variable context. The context must have one of the following three forms: 'A' 'A(J)' 'A(I,J)' where A is a format variable name and I and J are either integer constants or format variable names. The format variable names must be composed of no more than six letters or digits and the first character must be a letter. The format variables may be of any mode - floating-point numbers are automatically converted to integers, numbers in other modes are used as they are in storage.
USAGE IN MAD - All format variables must be names that exist in the program in which the I/O statement occurs whose format uses the format variables. I.e., dummy arguments cannot be used as format variables. All variables used as format variables must be so declared within the mad program. See the MAD MANUAL.

USAGE IN UMAP - See Appendix II of this write-up.
EXAMPLES:
(A) Values of X(I) can be plotted versus time (as represented by lines in the paper) as follows (assuming column 66 to represent 0):

```
      THROUGH QQ, FOR I=1,1,I.G.N
QQ    PRINT FORMAT OUT
      VECTOR VALUES OUT =$T66,S'X(I)',1H**$
      FORMAT VARIABLE X,I
```

(B) The following are legitimate (although highly improbable):
'SCALE' P 'NBR' F 'WIDTH' . 'DECDIG' *
'A(1)' P 'A(2)' E 'A(3)' . 'A(4)', 'A(5)' B 'A(6)' I 'A(7)' *
(C) Usage as switches:
H=THE TRIANGLE IS=,'SWITCH'(H* NOT*),
H= A RIGHT TRIANGLE=*
and assuming SW1=.NOT.SW2, the following might be used:
'SW1'(5E)'SW2(5F)15.5
Note that the multiplicity for the E or F must be inside the parentheses to keep it separate from the parenthesis multiplicity represented by SW1 or SW2.
NOTE - When using a format variable for the multiplicity, remember that varying the multiplicity does not vary the number of items on the list. If it is necessary to skip list items, use data-transmission format terms with zero field width to do it.

D. FORMAT-OFF CONTEXT.
A multiplicity of zero in front of a format term or left parenthesis means do it zero times, i.e., do not do it. Therefore, 0F10.5 will do nothing, and S10,0(S10,F10.5,3HTRA/),I3* will skip 10 and print

I/O SUBROUTINE (CONTINUED)

out a 3 column integer. Nothing inside the parentheses will be done. (This finds most use when there is a format variable context, rather than an explicit zero multiplicity, in front of the left parenthesis.) A zero multiplicity in front of a left parenthesis causes a change from normal context to format-off context. When scanning in this context, the only things recognized are left and right parentheses. The format terminator is not recognized. The context changes back to normal context when the right parenthesis that matches the left parenthesis which had the zero multiplicity is found.

VI. CARRIAGE CONTROL.

The first column of every print line is treated differently than the other columns by the computer that does the off-line printing. The first column is inspected and if it is a legal carriage control (see table on page 3.6-10 of this manual), the printer carriage is positioned according to the carriage control, the first column is blanked out, and the line is then printed. If the first column is not a legal carriage control, the printer single spaces, and then the entire line is printed out.

VII. STQUO.

There is another entry point to the general conversion subroutine, .IOH, which the user may call on directly. Since this is very closely tied in with the format scan, it is presented here, rather than as a separate subroutine. (Still another entry point, IOHSIZ, is described at page 3.8-34 of this manual.)

PURPOSE: This allows starting an I/O statement without reading in a new card (input) or blanking out the line-image (output). The line-pointer is left where it was at the conclusion of the last I/O call. This effect occurs only on the first I/O call after each call on STQUO. On subsequent I/O calls, things are reset, as normal. This subroutine is usually used in conjunction with the N modifier.

CALLING SEQUENCES:

```
MAD          EXECUTE STQUO.
UMAP         CALL STQUO
```

NOTE: When STQUO is used, the .IOH control information in erasable (see Appendix III of this write-up) must be left undisturbed between I/O calls.

APPENDIX I

ERROR COMMENTS

The normal procedure after detection of an error is to print a description of the error along with other pertinent information and then terminate execution of the program. This procedure can be altered by executing the subroutine SETERR prior to the I/O statement. See the write-up on SETERR. The table below gives the error comments and their associated error numbers. The error numbers are increased by 100 if the error occurs during output (e.g., error number =1 if an illegal character appears in the format during input, and error number =101 if an illegal character appears in the format during output). In each of the error comments below, X's stand for characters or numbers that are filled in.

ERROR NUMBER	ERROR COMMENT
1	ILLEGAL CHARACTER IN FORMAT OFFENDING CHARACTER IS 'X' (XX OCTAL)
2	FORMAT SPECIFIES MORE THAN XXX COLUMNS (WHERE XXX IS THE APPROPRIATE NUMBER FOR THE TYPE OF INPUT/OUTPUT)
3	ILLEGAL CHARACTER ON CARD IN COLUMN XXX OFFENDING CHARACTER IS 'X' (XX OCTAL)
4	NUMBER NOT IN MACHINE RANGE
5	NUMBER EXCEEDS SPECIFIED FIELD WIDTH. NUMBER IS XXXXXXXXXXXX (OCTAL) SAME NUMBER, CONVERTED ACCORDING TO FORMAT, IS (NUMBER IS PRINTED OUT AS CONVERTED) THE NUMBER CAME FROM LOCATION XXXXX VIA THE STR AT LOCATION XXXXX
6	ILLEGAL BCD TAPE NUMBER or ILLEGAL BINARY TAPE NUMBER
7	BINARY CARD IN BCD DATA or BCD CARD IN BINARY DATA
8	BAD BINARY BLOCK
9	TOO MANY BINARY BLOCKS
10	BASE FOR CONVERSION IS LESS THAN 2 OR GREATER THAN 19
11	MORE RIGHT PARENS THAN LEFT PARENS
12	NUMBER EXCEEDS SPECIFIED FIELD WIDTH (NORMALLY ERROR 5 WILL OCCUR. IF THIS ONE OCCURS IT MEANS BUFFER LENGTH WAS EXCEEDED.)
13	THE NAME XXXXXX IS NOT A FORMAT VARIABLE IN THE CALLING PROGRAM
14	DIMENSIONING OR SUBSCRIPTING ERROR
15	MULTIPLE CONVERSIONS SPECIFIED IN FORMAT TERM, PROBABLY DUE TO MISSING COMMA OR BAD HOLLERITH COUNT
16	LIST NON-EMPTY BUT FORMAT SPECIFIES NO CONVERSIONS
17	FORMAT WORD IS CORE CONSTANT

APPENDIX I - ERROR COMMENTS (CONTINUED)

Error comments 6 through 9 are generated by the BCD and binary tape subroutines. The others are generated by .IOH. Each of the above error comments generated by .IOH is followed by:

```
ERROR FOUND WHILE PROCESSING FORMAT WORD 'XXXXXX'

WHICH OCCURRED AT LOCATION XXXXX
        OUTPUT LINE
PRESENT     or      IMAGE IS
         INPUT CARD
          (LINE-IMAGE PRINTED HERE)

I/O STATEMENT BEGINS AT LOCATION XXXXX.
```

APPENDIX II

FORMAT VARIABLES AND SIMPLE I/O IN UMAP

Both use of format variables and use of the simple I/O routines require a symbol table which contains all variables that will be used as format variables or which will be referred to in simple I/O. The location of the symbol table that is given in the subroutine call is the location of a backwards-stored two entry table. The base location of this table is a word containing the length of the table. The structure of the table entries is:

```
LOCATION I      BCD name of variable.
LOCATION I+1    ADDRESS - location of variable.
                TAG     - Mode of variable.
                DECREMENT - Location of dimension vector
                            for this variable.  (0 if
                            there is no dimension vector).
                PREFIX - +0 if variable is not subscriptable.
                         -0 if the variable is subscriptable.
```

The modes are:

```
0    Floating point
1    Integer
2    Boolean
3    Function name
4    Statement label.
```

If the variable is going to be referred to with a double subscript, a dimension vector, as described in the MAD MANUAL, must be provided. A simple example of a symbol table would be

```
        BCI 1,SWITCH
        PZE SWITCH,2
STLOC   PZE 2
```

and an I/O call using it

```
        CALLIO .PRINT
        FMT    STLOC,,FORM
        ENDIO
```

APPENDIX II - FORMAT VARIABLES AND SIMPLE I/O IN UMAP (CONTINUED)

Where the format is

```
        FORM BCI  *,H=A IS=,'SWITCH'(H* NOT*),H= TRUE=*
```

A larger example will be given later.

SIMPLE I/O: The UMAP calling sequences to do the equivalent of the MAD simple I/O statements are:

```
        READ DATA -    CALL .RDATA,STLOC
        READ AND PRINT DATA - CALL .RPDTA,STLOC
        PRINT RESULTS - CALL .PRSLT,STLOC,LIST,0
        PRINT BCD RESULTS - CALL .PRBCD,STLOC,LIST,0
        PRINT OCTAL RESULTS - CALL .PROCT,STLOC,LIST,0
        PRINT COMMENT - USE THE SUBROUTINE SPRINT
```

WHERE STLOC is the location of the symbol table.
LIST is a normal subroutine list (single parameters or blocks), not an I/O list. (I.e., 'BLK's or 'PAR's, not 'IOP's.)

EXAMPLES: The first example reads in an M by N matrix, transposes it, and prints it out.

```
    $ASSEMBLE,EXECUTE,DUMP
    AGAIN  CALL    .RDATA,STLOC
           CALL    .PRSLT,STLOC,N,M,0
           LDQ     N
           MPY     M
           XCA
           SBM     L1
           SUB     =1
           ALS     18
           STD     L1
           CALL    TRANS1,MATRIX-1,M,N,TEMP
           PRINT   FMT,LABEL,...,LABEL-22,0
           PRINT   STLOC,...,FMT2
    L1     IOP     MATRIX-1,,**
           ENDIO
           TRA     AGAIN
    *
    FMT    BCI     *,22C6*
    FMT2   BCI     *,1HO,'M'F6.1*
    TEMP   BTS     15
    LABEL  BTS     30
    MATRIX BTS     500
    N      PZE
    *
    M      PZE     **            DIMENSION VECTOR FOR MATRIX
           PZE     1
    MDIM   PZE     2
    *
    FLOAT  EQU     0             DEFINE MODES
    INTGR  EQU     1
    STBEG  BCI     1,N           SYMBOL TABLE
           PZE     N,INTGR
           BCI     1,M
           PZE     M,INTGR
           BCI     1,MATRIX
```

APPENDIX II - FORMAT VARIABLES AND SIMPLE I/O IN UMAP (CONTINUED)

```
        MZE     MATRIX,FLOAT,MDIM
        BCI     1,LABEL
        MZE     LABEL,INTGR
STLOC   PZE     STLOC-STBEG
*
        END
$DATA
LABEL=$1TITLE
$
M=3, N=2, MATRIX(1,1)=1,2,3,4,5,6   *
```

The second example reads in N,M, a vector, and the length of the vector, and then prints out the Nth through Mth elements of the vector. It illustrates the use of zero field width to get rid of list elements. It also shows the use of a macro to make the symbol table construction easier.

```
$ASSEMBLE, EXECUTE, DUMP
AGAIN   CALL    .RDATA,STLOC        GET N,M,LENGTH,VECTOR
        STZ     SK2
        CLA     N
        SUB     =1
        STO     SK1
        SUB     M
        STA     SK2
        CLA     LENGTH
        SUB     M
        STO     SK3
        LAC     LENGTH,1
        TXI     *+1,1,VECTOR
        SXD     L1,1
        PRINT   STLOC,...,FMT
L1      IOP     VECTOR-1,,**
        ENDIO
        TRA     AGAIN
*
FMT     BCI     *,1HO,'SK1'F0,'SK2'F5,'SK3'F0*
        ASSIGN  N,M,LENGTH,SK1,SK2,SK3
VECTOR  BTS     20
*
SYMTAB  MACRO   NAME,SBBL,MODE,DIMV
        BCI     1,NAME
        IFF     0,/CRS/DIMV
        SBBL    NAME,MODE,DIMV
        IFF     1,/CRS/DIMV
        SBBL    NAME,MODE
SYMTAB  END
*
FLOAT   EQU     0
INTGR   EQU     1
STBEG   SYMTAB  N,PZE,INTGR
```

APPENDIX II - FORMAT VARIABLES AND SIMPLE I/O IN UMAP (CONTINUED)

```
        SYMTAB  M,PZE,INTGR
        SYMTAB  LENGTH,PZE,INTGR
        SYMTAB  VECTOR,MZE,FLOAT
        SYMTAB  SK1,PZE,INTGR
        SYMTAB  SK2,PZE,INTGR
        SYMTAB  SK3,PZE,INTGR
STLOC   PZE     STLOC-STBEG
*
        END
$DATA
N=2,M=5,LENGTH=8,VECTOR(1) = 1,2,3,4,5,6,7,8    *
N=1, M=5, LENGTH=5, VECTOR(1) = 1,2,3,4,5    *
```

APPENDIX III

STORAGE USAGE OF .IOH

To save space for the user temporary storage, buffers, the line-image, and a communication region with the other I/O subroutines are kept in low core, between 7300_8 and 7777_8. Control information and the push-down stack for parentheses are kept in erasable. The control information is as follows (all locations are in octal):

LOCATION	CONTENTS (ALWAYS INTEGER)
77752	MULTIPLICITY
77751	CURRENT COUNT (THIS IS THE COUNT THAT IS BEING ACCUMULATED DURING THE FORMAT SCAN)
77750	BASE FOR CONVERSION
77747	PRECISION (INTERNALLY SET COUNTER USED DURING CONVERSION)
77746	SCALE FACTOR
77745	LINE POINTER
77744	PUSH-DOWN POINTER (CONTAINS NUMBER OF WORDS OF PUSH-DOWN STACK BEING USED)
77743	FIELD WIDTH
77742	BASE (HIGH-ORDER END) OF THE PUSH-DOWN STACK.

The locations 77753 to 77777 are not disturbed.

The above storage allocations, being dependent on the internal structure of .IOH, are capable of changing at future dates.

At any given time, these cells hold the current control information for the format term being processed. Since this control information must be preserved during an I/O call, any function whose call is imbedded in an I/O list must not disturb this section of erasable. Only functions, not subroutines, occur in I/O lists.

APPENDIX IV

A BRIEF TABLE OF THE CHARACTERS IN NORMAL CONTEXT

TYPE	CHARACTERS
Control	
Data-transmission	A C E F G I K O
Non-data-transmission	H / * S T X
Modifying	B D L M N P R V W Z
Break	( ) , *
Change of context	0 IN FRONT OF (
	'
	H

APPENDIX V

A BRIEF TABLE OF APPLICABLE MODIFIERS

CONTROL CHARACTER	MODIFIERS
A,C	R W Z
E	D L M P V W
F	D L M P V W
G	D L M P V W
I	B L M V W
K,O	L W
H	
/	N
*	N
S	
T	
X	

APPENDIX VI

LISTING OF OTHER SUBROUTINES IN MAMOS MANUAL

EXPONIENTIATION SUBROUTINES

MAD automatically inserts calls to these in programs it generates when it finds a '.P.' in the source program.

A. .01311 Integer base, integer exponent
B. .01301 Floating base, integer exponent
C. .01300 Floating base, floating exponent

Calling sequence for the above 3 subroutines is TSX XXXXXX,4 (where XXXXXX is the name of the subroutine) with the base in the accumulator and the exponent in the MQ. Result is in the accumulator on return, mode of result is mode of base.

INPUT-OUTPUT SUBROUTINES

I. BCD (I/O with conversion via format)

A. DESCRIPTION.

The user, or the translator under the user's direction, calls directly on a preliminary routine (e.g., .PRINT). Each preliminary routine puts the following information in 4 special control locations in low core and then calls the conversion routine (.IOH) -- max number of columns, location of format information, direction of format information storage, location of a routine to call to do the input or output, location of the symbol table (if any), whether on-line or off-line, whether input or output. When the conversion routine needs a card image or has a line image ready to go out, it transfers to the location given it in the control information, which is the location of a 'connecting routine'. This 'connecting routine' may (e.g., .WR.) or may not (e.g., WR) be an actual subroutine in the sense that its name is in the library dictionary. (It is actually another, but separate, section of the preliminary routine.) The 'connecting routine' sets things up and then calls on a low-core 'transmission' subroutine (e.g., SPRINT) to do the actual tape writing or reading. In the list below, the four routines are listed in order for each function - preliminary, conversion, connecting, and transmission (e.g., .PRINT -- .IOH -- .WR. -- SPRINT).

B. INPUT - from system input tape (7).

1. Called by 'READ FORMAT' in MAD, or direct user call of .READ (or equivalent) in UMAP.
   .READ -- .IOH -- .RD. -- SCARDS
2. Called by 'LOOK AT FORMAT' in MAD, or direct user call of .LOOK (or equivalent) in UMAP.
   .LOOK -- .IOH -- .RD. -- SPEEK

C. INPUT - Arbitrary tape.

1. Called by 'READ BCD TAPE' in MAD, or direct user call of .TAPRD (or equivalent) in UMAP.
   .TAPRD -- .IOH -- .RD. -- SCARDS (If tape NBR = 7)
   .TAPRD -- .IOH -- RD -- RDSDEC (Otherwise)

APPENDIX VI - LISTING OF OTHER SUBROUTINES IN MAMOS MANUAL (CONTINUED)

D. PRINT OUTPUT - On system output tape (6).
   1. Called by 'PRINT FORMAT' in MAD, or direct user call of .PRINT (or equivalent) in UMAP.
      .PRINT -- .IOH -- .WR. -- SPRINT
E. PUNCH OUTPUT - On system output tape (5).
   1. Called by 'PUNCH FORMAT' in MAD, or direct user call of
      .PUNCH -- .IOH -- .PC. -- DPUNCH
F. OUTPUT - Arbitrary tape.
   1. Called by 'WRITE BCD TAPE' in MAD, or direct user call of .TAPWR (or equivalent) in UMAP.
      .TAPWR -- .IOH -- .WR. -- SPRINT (If tape NBR = 6)
      .TAPWR -- .IOH -- .PC. -- DPUNCH (If tape NBR = 5)
      .TAPWR -- .IOH -- WR -- WRSDEC (Otherwise)
G. PRINT ON LINE.
   1. Called by 'PRINT ON LINE FORMAT' in MAD, or direct user call of .COMNT (or equivalent) in UMAP.
      .COMNT -- .IOH -- PR -- ONLINE
          (and .IOH calls on SKIP6 at end)
H. SIMPLE INPUT/OUTPUT ROUTINES.
   1. Called by 'READ DATA' in MAD
      .RDATA -- SCARDS
      (.RDATA does its own conversion and, hence, calls only on the low-core subroutine SCARDS for the card images.)
   2. Called by 'READ AND PRINT DATA' in MAD
      .RPDTA -- SCARDS
      (.RPDTA is another entry to .RDATA)
   3. Called by 'PRINT COMMENT' in MAD
      .PCOMT -- SPRINT
      (In this case, no conversion is necessary, so the low-core subroutine SPRINT is called directly.)
   4. Called by 'PRINT RESULTS' in MAD
      .PRSLT -- .PRINT -- .IOH -- .WR. -- SPRINT
      (.PRSLT sets up format and list and calls on .PRINT, and then the sequence is the same as for .PRINT)
   5. Called by 'PRINT BCD RESULTS' in MAD
      .PRBCD -- .PRINT -- .IOH -- .WR. -- SPRINT
      (.PRBCD is another entry to .PRSLT)
   6. Called by 'PRINT OCTAL RESULTS' in MAD
      .PROCT -- .PRINT -- .IOH -- .WR. -- SPRINT
      (.PROCT is another entry to .PRSLT)
I. GENERAL AUXILIARY ROUTINES.
   1. SKIP6 Causes online printer to skip to next sixth of a page.
   2. DBLSPC Causes online printer to double-space.

II. BINARY (I/O WITHOUT CONVERSION)
A. DESCRIPTION.
   1. GENERAL.
      The MAD translator, under the user's direction, calls directly on a combined preliminary and select routine (e.g., .RBIN). The purpose of this routine is to preset certain instructions involving tape selects. A check is also made to see if the tape requested is a legal tape and, if so, if it is a system I/O tape. In the latter case, calls for I/O buffer tape routines (SCARDS and SPUNCH) are substituted for selects normally used on scratch tapes. In these

APPENDIX VI - LISTING OF OTHER SUBROUTINES IN MAMOS MANUAL (CONTINUED)

cases the I/O list may not specify more than 28 words, of which only 26-2/3 may be effectively read or punched.

Control is then transferred to an I/O list processor routine. In MAD, the preliminary routine calls a physically separate subroutine, .IOB, which returns to the preliminary routine which gives the actual select. The I/O lists themselves are identical to those of equivalent BCD tape routines.

2. MAD.

When writing scratch tapes in MAD, every element of the I/O list is converted into a channel command to transmit the specified location(s) to or from tape. A vector of these is built up in an area of low core. The maximum length of this vector is 200. Thus, a MAD binary tape statement may not generate more than 200 separate list items. An attempt to process a list generating more than 200 channel commands will result in the error comment 'TOO MANY BINARY BLOCKS', and execution will be terminated. Because a data channel transmits successive words in order of increasing address, a MAD block I/O list element (e.g., V(1)...V(10)) will always generate a channel command such that the word with the lowest address (i.e., the highest subscript) is transmitted first (e.g., IOBP V-10,,10). To try to minimize accidental errors, 'REVERSE ORDER' blocks (e.g., V(10)...V(1) are considered illegal by .IOB. An attempt to process one will produce the error comment 'BAD BINARY BLOCK', and execution will be terminated. Care should be used when writing and reading scratch tapes with differently structured I/O lists, since any blocks will be transmitted 'BACKWARDS' from the order stated (e.g., V(10), then V(9), etc., through V(1), not V(1) through V(10)). When a list terminator (STR 0,,0) is recognized, the last channel command is changed to an IORT. A return is then made to the calling preliminary routine which selects the tape through the direct binary select routines, using the vector of channel commands just generated. Then a delay is initiated to wait until tape transmission has been completed, at which time control is returned to the MAD program. Thus each MAD binary tape statement will process one physical record on tape.

If transmission is occurring on a system I/O tape .IOB is not called. Instead, a simple list processor, like that of .IOH, which is contained in the preliminary routine, is used to process the card image read or the card image to be punched via the system I/O buffer routines.

Below are the subroutines called by binary tape statements in MAD. In every case, the left-most name is that of the preliminary and select routine, and the right-most name is that of a low core routine, either a buffer routine for system I/O tapes or a direct select routine used with scratch tapes.

B. INPUT - From system input tape (7).

1. Called by 'READ BINARY TAPE 7' in MAD

.RBIN -- SCARDS

APPENDIX VI - LISTING OF OTHER SUBROUTINES IN MAMOS MANUAL (CONTINUED)

C. INPUT - From arbitrary tape
   1. Called by 'READ BINARY TAPE' in MAD
      .RBIN -- .IOB -- RDSBIN
D. PUNCH OUTPUT - On system peripheral punch tape (5).
   1. Called by 'WRITE BINARY TAPE 5' in MAD
      .WBIN -- SPUNCH
E. OUTPUT - Arbitrary tape.
   1. Called by 'WRITE BINARY TAPE' in MAD
      .WBIN -- .IOB -- WRSBIN

LIST MANIPULATION SUBROUTINES

I. Called by 'SET LIST TO' in MAD, 'SETTO' pseudo-op in UMAP, or direct subroutine call:
   .SET

II. Called by 'SAVE DATA' in MAD, 'SAVE' pseudo-op in UMAP, or direct subroutine call:
   .SAVE

III. Called by 'SAVE RETURN' in MAD, or direct subroutine call:
   .SAVRN

IV. Called by 'RESTORE DATA' in MAD, 'RESTOR' pseudo-op in UMAP, or direct subroutine call:
   .RSTOR

V. Called by 'RESTORE RETURN' in MAD, or direct subroutine call:
   .RSTRN

.SET, .SAVE, and .RSTOR setup and manipulate push-down type lists. .SET has one argument which is the first element of the array to be used as the list. This element must be preset to the initial list length, and thereafter will have in it the current list length. .SAVE and .RSTOR have I/O calling sequences which terminate with blank IOP instructions. .SAVE puts elements on the specified list in the order that they occur in the calling sequence. .RSTOR obtains elements from the end of the list and puts them in the locations specified by the calling sequence and in the order that they occur in the calling sequence. Any computation desired may occur within the calling sequence. .SAVRN and .RSTRN are similar to .SAVE and .RSTOR. They are used to save and restore the prolog of the MAD function they are called from.

PAUSE SUBROUTINE

Called by 'PAUSE NO.' statement in MAD, and 'PAUSE' pseudo-op in UMAP. Causes the machine to stop in execution in such a manner that the program may be continued by pressing the start button. Time during the pause is counted as processing time, not execution time and, hence, does not count against the user's execution time estimate. All high speed registers are preserved.

APPENDIX VI - LISTING OF OTHER SUBROUTINES IN MAMOS MANUAL (CONTINUED)

SUBSCRIPTION SUBROUTINES

I. .03311 and .03310 (two names for same routine).
When MAD finds a 2 dimensional subscript, A(I,J), it calls on this subroutine to find the linear subscript, R, that corresponds to it. (I.e., A(R) and A(I,J) refer to the same element)
CALLING SEQUENCE IS

```
CLA  I
LDQ  J
TSX  .03311,4
BLK  A,,ADIM
```

where ADIM is the name of the dimension vector for A. R is in the accumulator on return.

II. MTX.
When MAD finds a 3-or-higher dimensional subscript, A(I,J,K,...), it calls on .MTX to find the corresponding linear subscript.
CALLING SEQUENCE IS

```
TSX  .MTX,4
BLK  A,,ADIM
PAR  I
PAR  J
PAR  K
 .    .
 .    .
 .    .
```

linear subscript is in the accumulator on return.

III. SYMM
.SUBS
Special subscription subroutines called by MAD when the special subscripting option is used.

NOTE: Since these are called by MAD, they assume MAD-type storage of vectors.

TAPE MANIPULATION ROUTINES

I. BACKSPACE RECORD.
A. Called by 'BACKSPACE RECORD OF TAPE' in MAD
.BSR

II. BACKSPACE FILE.
A. Called by 'BACKSPACE FILE OF TAPE' in MAD
.BSF

III. REWIND.
A. Called by 'REWIND TAPE' in MAD
.RWT

IV. REWIND AND UNLOAD.
A. Called by 'UNLOAD TAPE' in MAD
.RUN

V. WRITE END-OF-FILE.
A. Called by 'END OF FILE TAPE' in MAD
.EFT

VI. SET DENSITY.
A. Called by 'SET LOW DENSITY TAPE' in MAD
.SETLO
B. Called by 'SET HIGH DENSITY TAPE' in MAD
.SETHI

INDEX TO MAMOS SUBROUTINE LIBRARY - BY ENTRY POINTS

ENTRY POINT NAMES	$LENGTH_8$	$ERASABLE\ USED_8$	PAGE
DCOMPZ, DZCOMP	41	0	3.8-15
DFAD, DFSB, DFMP, DFDP, SFDP, DCEXIT	147	0	-16
DISMNT	143	0	-15
DPFA	14	1	-17
DPFDV	120	4	-17
DPFM	23	3	-17
DPMAT	102	317	-18
DSLE1, DSLE2	1207	146**	-19
DSQRT	37	0	-18
.EFT, (EFT)	14	0	-113
EIGN	620	16	-21
ELOG	102	4	-22
ERF	153	6	-25
ERRFN, FREQ	105	4	-25
.EXIT	55	0	-23
EXP	113	5	-24
EXP1	4	0	-26
EXP2	4	0	-26
EXP3	4	0	-27
FSPILL, RSPILL	60	0	-27
GAMMA	66	2	-28
GAUSS	145	145**	-29
GJRDT	216	317	-30
GJRDTP	210	317	-31
HAS1, HAS1S	516	50	-32

ENTRY POINT NAMES	$LENGTH_8$	$ERASABLE\ USED_8$	PAGE
IBDS	446	234**	3.8-33
IEF1	236	17	-33
.IOB	127	0	-110
.IOH, STQUO, (FIL), (RTN), IOHSIZ	4147	0	-85
IOHSIZ	See .IOH ROUTINE		-34
ITINT	362	1	-35
.LOOK, .READ, .RD., (CSH)	147	0	-109
LSH, RSH	13	0	-39
MOUNT, LABEL	1373	0	-39,40
MOVER	113	1	-41
MTX	33	0	-113
NASQ	105	0	-41
NDRN1A, NDRN1B, NDRN1C, NDRN1D	171	3	-42
OFFTRC, ONTRC	11	0	-43
PCPCH	222	0	-43
.PCOMT	17	0	-110
PLOT1, PLOT2, PLOT3, PLOT4, OMIT, FPLOT4	1777	0	-44
.PRINT, .WR.	64	0	-110
.PRSLT, .PRBCD, .PROCT	1007	0	-110
.PUNCH, .PC., (SCH)	0	0	-110
RAM2A, RAM2B, RAM2C, RAM2D	64	1	-57
RANDND	73	0	-59
RANDOM	30	0	-58
.RBIN	234	0	-111
.RDATA, .RPDATA	1102	0	-110

ENTRY POINT NAMES	$LENGTH_8$	$ERASABLE\ USED_8$	PAGE
REPLCE	31	3	3.8-60
RKDEQ, SETRKD	141	1	-61
.RUN, .RWT, (RWT)	25	0	-113
SAVCOR	165	4	-62
.SAVRN, .SAVE, .RSTRN, .RSTOR, .SET	213	0	-112
SELPGM, SEQPGM	35	1	-63
SET2, SET8	14	0	-64
SETEOF, SETEFL	24	0	-64
SETERR, .ERR	66	0	-65
SETETT	25	0	-66
SETFTP, RSTFTP	37	0	-66
.SETLO, .SETHI	21	0	-113
SETPLT, USTPLT	701	4064	-67
SETUP, CONVRT, ADD, SUB, MPY, DIV, RMDR, RECNVT, IF	1333	320	-80
SIN, COS	177	4	-70
SKIP	33	0	-70
SLEC, SLEG, SLEM	110	1	-71
SPREAD, GATHER, FSPREAD, FGATHR	154	0	-72
SQRT	57	2	-73
(STH), .TAPWR	207	0	-110
.SUBS, .0311, .0310	42	0	-113
SYMM	41	0	-113
TAB	307	22	-74
TANH	136	5	-73
TRANS	66	5	-75
TRANS1	116	4	-75

ENTRY POINT NAMES	$LENGTH_8$	$ERASABLE\ USED_8$	PAGE
(TSH), .TAPRD	215	0	3.8-109
UITR1, UITR1A	164	0	-77
UITR2, UITR2A	322	0	-78
UITR3, UITR3A	304	0	-79
.WBIN	211	0	-112
ZER2, ZER3, ZER4, ZER5, ZER6	670	25	-82
ZERO, SPRAY	34	0	-84
.01300	66	11	-109
.01301	76	3	-109
.01311	64	3	-109
.03311, .03310	15	0	-113

**This routine uses a variable amount of erasable storage. The number given is the minimum erasable needed.

INDEX TO MAMOS SUBROUTINE LIBRARY - BY FUNCTION