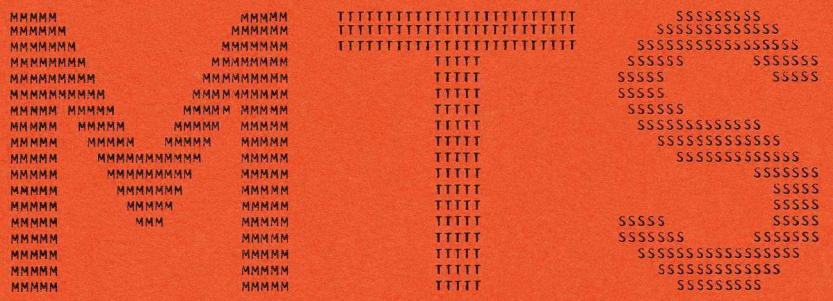
--------999242----------999242----------999242----------999242----------999242----------999242-------

JOB NO. 999242 UNIVERSITY OF MICHIGAN TERMINAL SYSTEM (MODEL CT137) 22:23.39 12-21-67



--------999242----------999242----------999242----------999242----------999242----------999242-------

$SIGNON QQSV
ILLEGAL SIGNON I.D.

VOLUME I

second edition
December 1, 1967

M T S

MICHIGAN TERMINAL SYSTEM

SECOND EDITION

December 1, 1967

This edition is a major revision of, and obsoletes, the first edition (June 1, 1967).

This work was made possible in part by support extended to the University of Michigan by the Advanced Research Projects Agency of the Department of Defense (contract number DA-49-083 OSA-3050 ARPA order number 716 administered through the Office of Research Administration, Ann Arbor).

DISCLAIMER

These MTS writeups are intended to represent the current state-of-the-system. This publication must not be construed as an obligation to maintain the system as so stated. The system is still being developed and additions, extensions, changes, and deletions will occur. At times these changes may result in some parts of the new system being incompatible with their corresponding parts of the old system.

Preface to the Second Edition

The first edition was published in June. In September we began collecting corrections for issuing a set of changed pages and discovered that we would be replacing two-thirds of the pages in the manual. Hence this new edition.

In an attempt to make things easier for people acquainted with the first edition, revision bars have been placed in the margin alongside of changed lines. If a section is completely replaced or is new, a revision bar is placed alongside of the section number only.

To keep the sizes manageable and to logically split the manual, a new major section (MTS-500) was created and made Volume II. This section is comprised of the IOH/360 and UMIST (change in name only) writeups, the bulk of the Fortran G (*FORTRAN) and assembler (*ASMBLR) writeups, and four new writeups. Also the two sections of subroutine writeups (253 and 254) were combined into one section of subroutine descriptions (253). In addition, a list of major differences from the first edition follows:

New Sections

Introductory and Usage Writeups
- Calling Sequences
- Data Concentrator User's Guide
- General view of UMMPS and MTS
- 1050 User's Guide
- 2250 User's Guide

Language Processors
- PIL writeup
- SNOBOL4 writeup
- WATFOR writeup
- 8ASS writeup

Library File Descriptions

*ASA	*ASMEDIT	*BATCH
*CATALOG	*CONVSNOBOL	*DOUBLE
*FILEDUMP	*FILESCAN	*GPAKDRAW
*GPAKGRID	*GPAKLIB	*GRAPHLIB
*GRAPHMAC	*IHC	*LINPG
*OSMAC	*PAL8SS	*PIL
*SDS	*SNOBOL4	*SQUASH
*SSP	*UPDATE	*WATERR
*WATFOR	*WATLIB	*2250EDIT
*8ASS	*8SSPAL	

Macro Descriptions

ACCEPT	BAS,BASR	DFAD,DFSB,DFMP
DFIX,EFIX	DISMOUNT	ENTER
EXIT	FLOAT	GETSPACE
MOUNT	SCARDS,SPRINT,SPUNCH,SERCOM,READ,WRITE	
SLT	SWPR	

Subroutine Descriptions

ATTNTRP	Bitwise Logical	Blocked I/O
EMPTY	E7090,D7090,E7090P,D7090P	
IOPMOD	LINPG	REWIND
REWIND#	SETIOERR	SETPFX

Extensively Changed Sections

Files and Devices (MTS-210)
IOH/360 (MTS-530)

Change of Name

UMIST (was TRAC)
LISTVTOC (was CATALOG)

Other major new features, in addition to the new library files, subroutines, macros and language processors listed above, are explicit and implicit concatenation of files (see MTS-210), ability of the user to specify a password (see descriptions of the $SET and $SIGNON commands), and the ability of a batch user to specify time, page, and card limits for his job (see Limits Specification, section MTS-225).

Donald W. Boettner
Computing Center
University of Michigan
Ann Arbor, Michigan
December 1, 1967

PREFACE

This manual represents an attempt to document, for the benefit of both users and the people working on it, a rapidly changing system. The system described here is essentially the system as it existed about June 1, 1967. By the time anyone reads this, various parts of this manual will most certainly be obsolete.

The Computing Center would like to acknowledge I.B.M. for sections, primarily listings of error comments, from their Fortran IV [G] Programmer's Guide and Assembler F Programmer's Guide which are reproduced here.

Acknowledgement should also be made to the numerous people who helped in preparation of this manual, especially Ronald Srodawa, for the Teletype User's Guide and Loader Description; Leonard Harding, Fortran writeup; Larry Flanigan, Concepts and Facilities; Charles (Kip) Moore, File Routines Internals; Fred Swartz, Batch User's Guide; Allan Emery, 2741 Users Guide and descriptions of debug commands and routines; Jay Jonekait, Tape User's Guide; Vic Streeter, Assembler writeup and plotting routines descriptions; Charlie Benet, for IOH writeup; Dave Mills, Data Concentrator User's Guide; Michael Alexander, proofreading and revision of the manual; and last but certainly not least, Karen Dymond, for keypunching this whole mess in TEXT90.

Donald W. Boettner
University of Michigan
Computing Center
June 1, 1967

VOLUME I

This comprises the first volume of the MTS manual. It contains writeups concerned with use of MTS and use of the language processors within MTS, whereas the second volume contains writeups concerned with the content of the languages available.

A complete table of contents which covers the entire manual follows.

MTS

(MICHIGAN TERMINAL SYSTEM)

"Writeups are like watches; the worst is better than none, and the best cannot be expected to go quite true."

[with apologies to Samuel Johnson]

GENERAL INTRODUCTION

MTS is a time-sharing system designed to be run primarily from remote terminals. It also has a batch mode to allow batch processing of jobs. The system allows the user at a remote terminal to create files consisting of general text or program material, to call in processors to translate these files, to run translated programs, to inspect and debug programs, to obtain hard copy of files on high speed printers and punches, to communicate between consoles, etc.

The system is described in a series of sections (of which this is the first), labelled as follows: The upper left hand corner of each page of a document contains

(1) the document number
(2) the date of issue

The document number is constructed as follows: The general form is

MTS - <section number> - <level number>

for example: MTS-210-0

Section 230 covers the commands (in alphabetical order). The section number for commands is extended by adding a slash followed by the first 3 letters of the command. If this is not unique, a colon and distinguishing letter(s) are added, and so on, e.g., MTS-230/CRE-0

Section 253 covers subroutines available to user programs (in alphabetical order). The section number is extended there by adding a slash followed by a five-digit number identifying the subroutine e.g., writeup on SPRINT has the number MTS-253/62515.

Section 255 covers macro library descriptions and is handled like section 253.

Section 280 covers library file descriptions and is handled like section 253.

USAGE DESCRIPTION

This major section of the manual is concerned with presenting a general overview of the system and usage guides for different types of users at different kinds of terminals.

CONCEPTS AND FACILITIES

A present day computing system is a complex blend of hardware (the actual physical components of a computer) and software (programs which control the functioning of the hardware). The purpose of the software is to provide to the user the full capabilities of the hardware without requiring a detailed knowledge of the hardware, and in addition to provide functions and facilities not directly available in the hardware. From the system viewpoint, on the other hand, the software provides for reasonably efficient usage of the hardware by removing the user from the basic levels of hardware function (such as I/O and interrupt handling). In effect, the user sees a computer which is the sum of the hardware and software facilities, so that he is, in fact, not running on the actual hardware computer which supports the software.

This manual is a description of the MTS (Michigan Terminal System); MTS is a software system designed to run on an IBM System 360 hardware configuration. This manual introduces the basic concepts and facilities of this software system; details and descriptions of the hardware must be obtained from the appropriate IBM manuals.

MTS is designed to handle both interactive operations (terminal operations) and batch operations. In interactive operations, the user is situated at a terminal device (such as a teletype) and communicates with the system in real time. Requests for system action are received, performed, and the results immediately returned to the user, who may then react and enter new requests. MTS is able to support several terminals at a given time by giving each user a certain amount of time in sequence. In batch operations, the user submits a deck to the Computing Center and waits until this deck has been run to obtain output. The center collects these decks into batches, and then runs a whole batch at once. Thus, turn-around-time is longer for batch operations; but due to the need of terminals and communication lines in interactive mode, batch operations are normally significantly cheaper for the same processing.

MTS contains a number of facilities to make the computer more readily accessible. The system contains an assembler (F level IBM assembler), two Fortran compilers (Fortran IV G level and WATFOR), an interactive calculator language (PIL), a symbol manipulation language (SNOBOL4), and a symbol-macro processor (UMIST). Additional translators will be available in the future. The system user is able to define, store, and use collections of information (files-see later) either from terminals or in batch. These collections are generally whatever the user wishes to file away for later use. Once defined, such collections may be modified, deleted, or reused at later times. The user from a terminal has full facilities for running, interrupting, modifying, and restarting (i.e., debugging) programs which he has written and translated. The system includes both a macro library for assembler users and a subroutine library

for all users, thus easing many of the programming burdens. In addition to these facilities, the user is able to communicate between terminals, to copy files and make hard copy of them, and, in general, to obtain all those services necessary to make the computer accessible as a tool.

This section will serve as a brief introduction to MTS. Basic definitions will be given, and simple examples will explain some of the functions of MTS. An understanding of the remainder of this section should allow one to use the remainder of this manual with relative ease.

A file is a logically ordered set of lines, where each line is some unit of information. For example, a line may be a source card of 80 characters, an object card, an output print line, or any other unit of information desired by the definor of the file. Files may be created, modified, or destroyed as necessary; they may be permanent or temporary for a single part of a job; they act as storage areas for collections of information which will be needed at some later date. There are three types of files in the system: user files, library files, and scratch files. Library files are files which are available to all users of the system; the translators are examples of such files. User files are private files defined by individual users and cataloged under particular ID numbers; such files are available only to those users who have the proper ID number. Scratch files are files created for use in a single job and destroyed automatically at the end of that job. All files in the system bear distinct names for reference purposes; in fact, a file must be named before it can be cataloged. A library file has a name which begins with an asterisk (*) and is followed by 1 to 15 characters; this name must be used for all references to the file. User files have names consisting of 1 to 12 characters; it is strongly suggested that only alphanumeric characters be used in naming user files, although other characters may work. To prevent different users from defining files of the same name, the system prefixes a four character ID to the file name given by the user; this ID is assigned to each user by the computing center and is unique to each user. A scratch file has a name which begins with a minus sign (-) and is followed by 1 to 8 characters.

Within a file, each line has a unique line number of the form sxxxxx.xxx, where s is a sign (plus or minus), x is a decimal digit, and the . is a period or decimal point. At the time a file is defined (created), one may supply with each line its line number; alternatively, one may simply supply to the system a beginning line number and an increment, and let the system supply a line number for each line. Either way, each line within a file has a unique line number. After a file has been created, various parts of the file may be referenced or modified by referring to the appropriate line numbers.

There is a set of special files in the system which may be used at any time and which are always defined. Each of these special files has a file name of the form *NAME*, where NAME specifies the particular special file. Thus, for example, *DUMMY* is a system special file which is always empty.

If one tries to read this file, an EOF (end of file) indication is received. If one attempts to place information in this file, the information is lost (an infinite garbage pail). This special file is useful during debugging or as an easy way to ignore some particular output. Other special files will be described later in this section.

Another type of name recognized by the system is a device name. This is the name of a particular physical device on the computer, and the built-in device names are always 4 characters in length. These names are used when one wishes to refer directly to some particular physical device (such as a card reader, a printer, a disk, etc.) rather than to a logical file. Under normal conditions, reference to device names would occur only during terminal operations. One difficulty does arise due to the presence of these device names; a user may unwittingly use such a name as a file name, causing problems in referencing the file. To prevent such problems, a pound sign (#) may be prefixed to a file name to indicate that it is a file name even if there is a device name of the same symbol.

There are a number of attributes or modifiers which may be attached to a file name wherever a file name occurs. An attribute is attached to a file name by prefixing the attribute with the character @ and placing the whole construction after the file name. For example, FILE@I is the file name FILE with the indexed (I) attribute. The various attributes which may be designated with a file name are specified elsewhere in this manual and will not be further discussed here.

Now let us consider how to run in the MTS system. The basic philosophy of MTS is that at any given time the system is either in command mode or some particular program is being run. When in command mode, there are two kinds of input lines to the system: command lines and data lines. A command line contains a $ in its first character position (column 1 if on a card) followed immediately by a command name, followed by whatever parameters, if any, are needed by the particular command. Any line which is not a command line is, by default, a data line. Command lines are used to provide commands to the system; that is, to provide information to the system, to request information from the system, or to specify some particular action to be taken by the system. Data lines are what the name implies: a data line is a line of data to put in a file. Whenever MTS finds a command line, the command is carried out. Whenever MTS finds a data line, the line is added to some current file for later processing; the particular file used here is whichever file the user is currently defining (the current active file).

On the other hand, when some particular program is being run, then all input lines are data for the program. The data required depends upon the program being run. If a translator is being run, then the data is a source program. If a loader is being run, then the data is an object module. If a user program is being run, then the data is whatever that program requires as input. Thus, a data line may be any one of the above possibilities.

To initiate a run (job) in MTS, the first line must be the command

```
$SIGNON id
```

where id is a four character ID assigned to the user. If this ID is acceptable, the user is signed on in the system and is ready to run. The ID given in this command is the ID used to prefix all user file names during the run. As part of the signon procedure, the system prints out the last time this ID was used to signon in the system. Furthermore, two special files are opened for this job; these are *SOURCE* and *SINK*. *SOURCE* is defined as the source of incoming lines; it is the terminal for terminal operations or the input device for batch. *SINK* is defined as the destination of outgoing lines; it is the terminal for terminal operations or the output device for batch. In addition, the special file *PUNCH* is defined as the punch device if this is a batch run. These special file names make it possible to refer to the pertinent devices without knowing their actual device names. Since the actual devices used for source, sink, and punch in batch runs varies from batch to batch, the use of these names is mandatory for references to the batch input and batch output devices.

To introduce some basic concepts and procedures in using MTS, let us now consider a simple use of the system. The examples in the remainder of this section will be written as batch input decks; the primary distinction between batch and terminal operations is that no interaction between the user and the system is possible in batch.

The following deck causes the compilation and execution of a Fortran program which computes square roots.

```
$SIGNON  IDNO
$RUN     *NEWFORT; 0=-LOAD
         NAMELIST /NL/SQ,ROOT
10       READ (5,NL)
         ROOT = SQRT(SQ)
         PRINT (6,N)
         GO TO 10
         END
$ENDFILE
$RUN     -LOAD  *SINK*; 5=*SOURCE* 6=*SINK*
 &NL   SQ=10. &END
 &NL   SQ=20. &END
$ENDFILE
$SIGNOFF
```

The signon statement has already been discussed. The next statement is the run command, and it requests that MTS run the program whose name is *NEWFORT. This is the Fortran compiler, so this statement effectively requests a compilation. (The various MTS commands are described in detail later in this manual. This section will not go into the full details of each command.) In order to understand the remainder of this command, it is necessary to define the concept of logical I/O unit. A logical I/O unit is a name which may be used in a program to request I/O (input/output) activity. The name is not connected to any particular device or file by

the program, so the program is independent of the actual device or file which eventually holds the information. The logical I/O units available in MTS are SCARDS, SPRINT, SERCOM, SPUNCH, and 0 through 9. If we look at the Fortran writeup in this manual, we see that Fortran expects a source deck on SCARDS and places a listing on SPRINT, error comments on SERCOM, and an object module on unit 0. Therefore, when a request is made to run Fortran, it is necessary to specify which devices or files are actually to be used for these purposes. The description of the run command shows that SCARDS is taken as the current source, if not specified; therefore we need not specify it since our program is in the source stream. Further, SPRINT and SERCOM are, by default, the current sink; again we need not specify these since we desire this output to return via the system output devices. Unit 0, however, has no default assignment; thus, it is necessary to assign some particular file to 0 to get an object deck for running. Since we want this file only long enough to collect an object deck for execution, we really need a temporary file which can be deleted at the end of the job. This may be done by specifying a file name preceded by a minus sign. MTS declares such a file as temporary and automatically destroys the file at the end of the job. Thus, on our run command we have 0=-LOAD. This tells MTS that a temporary file named -LOAD should be created, and all references to 0 in Fortran should refer to -LOAD. Therefore, we run Fortran with the source deck on *SOURCE*, the printed output on *SINK*, and our object module on -LOAD. In this particular case, we will have an unsuccessful compilation, since our PRINT statement has an illegal namelist name in it. Therefore, we will not have an execution. To correct this, we would change our erroneous statement to

```
PRINT  (6,NL)
```

and resubmit the deck. On this run, compilation will be successful. The $ENDFILE command simply signals to Fortran that it should quit reading the source file and finish the compilation. The next command requests running the object module produced by Fortran by requesting that -LOAD be run. The *SINK* requests that a loading map be placed on the output; a loading map gives a list of the various programs loaded into physical memory and their locations. This is handy for later debugging of the program. To provide input and output files for the program, 5 is defined as *SOURCE* and 6 as *SINK*, where 5 and 6 are the data set reference numbers used in the READ and PRINT statements. Note that these data set reference numbers are simply the logical I/O units defined earlier. Thus, our program expects to read from logical I/O unit 5, and when we run the program we must tell the system what file to use when 5 is referenced. In this case we chose *SOURCE* since our data will be there. The $ENDFILE after the data tells the program that this is the end of the data. The final command $SIGNOFF must be the physically last line of any run.

Now let us consider a slightly different version of the previous deck:

```
$SIGNON  IDNO
$CREATE SQRT
$NUMBER
        NAMELIST  /NL/SQ, ROOT
10      READ  (5,NL)
        ROOT = SQRT(SQ)
        PRINT  (6,N)
```

```
      GO TO 10
      END
$UNNUMBER
$LIST  SQRT
$RUN   *FORTRAN; 0=-LOAD SCARDS=SQRT
$RUN   -LOAD *SINK*; 5=*SOURCE* 6=*SINK*
 &NL   SQ=10.  &END
 &NL   SQ=20.  &END
$ENDFILE
$SIGNOFF
```

Here, rather than compiling our source deck from *SOURCE*, we first place it in a permanent user file. The $CREATE command requests that a user file with the name SQRT be created; it also makes this file (SQRT) the current active file. A special file named *AFD* is defined as the current active file and may be used for references; hence, SQRT and *AFD* each refer to the same file after this $CREATE command has been processed. The $NUMBER command indicates that the system should supply a line number for each data line which it encounters; it is possible to specify a beginning line number and an increment in this command if one wishes, but here we have taken the default case of starting at line number 1 with an increment of 1. Now, as the source program is read, each source program statement is a data line to the system. Since data lines are always stored in the current active file, the source program statements are successively numbered and placed in the file SQRT. (If the $NUMBER command had not been given, MTS would have searched each line for a line number and used it if there; otherwise, a line number of zero would have been assumed. Disastrous results would obviously have followed.) Note that no $ENDFILE command is needed here since the source deck is not being read by Fortran. The $UNNUMBER command turns off the automatic line numbering by MTS. The $LIST command requests that a listing of the file SQRT be placed on *SINK* (by default). This command would generate the following output:

```
1       NAMELIST  /NL/ SQ, ROOT
2   10  READ (5,NL)
3       ROOT = SQRT (SQ)
4       PRINT (6,N)
5       GO TO 10
6       END
```

The numbers on the left of each line are the line numbers from the file SQRT. The remainder of this deck is as before, except that now on the $RUN command for Fortran we must specify SCARDS=SQRT so Fortran will take its source deck from our new file. As before, this deck will not compile due to the erroneous PRINT statement. However, since the source deck is in a permanent user file, we need not read in the whole deck after removing our program error. We can simply correct our file and run. The following deck does this:

```
$SIGNON  IDNO
$GET  SQRT
4     PRINT (6,NL)
$RUN  *FORTRAN;....
```

.
.
.

where the dots indicate same as the previous deck. The $GET command makes SQRT the current active file. Note that we cannot use $CREATE again, since the system would balk at creating two files of the same name. Since $NUMBER was not given, MTS assumes the data line has its own line number. Thus, the 4 is stripped from the PRINT (6,NL) line, used as a line number, and our data line replaces line number 4 in SQRT; this corrects our source program in SQRT. If we had wanted to add statements to our file, we could have picked line numbers between those in the file; in this case the new lines would be inserted in the appropriate places. Thus, the following deck would place a comment card after the NAMELIST statement in SQRT:

```
$GET  SQRT
1.5C  SQRT  PROGRAM
```

There are a number of commands which have not been discussed here, but the reader should by now have gained some basic understanding of the concepts and procedures of MTS. It is possible to destroy or empty files, to change the source or sink assignments, etc. There are also commands which allow one to interrupt a running program, dump out pertinent variables from the program, modify various of the locations in the program, and then restart the program. These capabilities are primarily of use to those running from terminals; the debugging facility providing by this type of interaction should be obvious.

This has been a brief introduction to the concepts of MTS. However, if the definitions and procedures discussed here have been understood, one should have little difficulty in obtaining more detailed information from the remainder of this manual.

CALLING CONVENTIONS

INTRODUCTION

A calling convention is a very rigid specification of the sequence of instructions to be used by a program to transfer control to another program (usually referred to as a subroutine). It is very desirable although not always practical to set up only one set of conventions to be used by all programs no matter what language they are written in so that FORTRAN programs may call MAD programs and assembly language programs and so forth. In the MTS system the OS type I calling conventions have been adapted as the standard. A complete specification of these standards can be found in the IBM System/360 Operating System Publication, Supervisor and Data Management Services, Form C28-6646. This writeup shall try to bring out pertinent details of these calling conventions.

Throughout this discussion we will refer to the terms calling program, called program, save area, and calling sequence. The calling program is the program which is in control and wants to now call another program (subroutines). The called program is the program (subroutine) which the calling program wants to call. The save area is an area belonging to the calling program which the called program uses to save and later restore general purpose registers. The save area has a very rigid format and is discussed in more detail later on. A calling sequence is the actual sequence of machine instructions which perform the tasks as specified by the calling conventions.

The facilities that must be provided by the calling conventions are:

1. Establish addressability and transfer to the entry point.
2. Pass parameters on to the called program.
3. Pass results back to the calling program.
4. Save and restore general purpose and floating point registers.
5. Re-establish addressability and return to the calling program.
6. Pass a return code (error indication) back to the calling program so it knows how things went.

The remainder of this writeup will describe the OS type I calling conventions to show how they are used and how the facilities listed above are provided for.

REGISTER AND STORAGE VARIANTS OF TYPE I CALLS

The OS Type I calling conventions actually consist of two very similar calling conventions, referred to as OS (I) S Type calling conventions and OS (I) R Type calling conventions. The two differ only in the way parameters and results are passed between the calling and called programs. The R refers to register and the S to storage.

The OS (I) R type calling conventions utilize the general purpose registers 0 and 1 for passing parameters and results. This allows only two parameters or results and cannot be generated in higher level languages as FORTRAN. Its advantages are that calling sequences are shorter and take less time to set up. These are very popular in lower-level system subroutines such as GETSPACE or GETFD.

The OS (I) S Type calling conventions require a pointer to a vector of address constants called a parameter list (in register 1). Since the parameter list can be of any required length, several parameters can be passed using OS (I) S Type calling convention. These conventions are used by system subroutines such as SCARDS or LINK and are generated by all function or subprogram references in FORTRAN. Results can be passed back by giving variables in the parameter list new values or via register 0.

PARAMETER LISTS

As stated above a parameter list is a vector of address constants. The parameter list must be on a full-word boundary and the entries are each four bytes long. The address of the first parameter is the first word of the list, the address of the second parameter the second word of the list, and so on. For example the parameter list for the FORTRAN statement

```
CALL QQSV(X,Y,Z)
```

might be written in assembly code as:

```
PAR  DC   A(X)      address of X
     DC   A(Y)      address of Y
     DC   A(Z)      address of Z
```

Now this parameter list works well enough when the parameter list for the subroutine is of fixed length, but there is not enough information yet to allow a subroutine to determine the length of the parameter list and hence accept variable length parameter lists. For this reason there are two types of parameter lists, fixed length parameter lists as described above, and an extended form of parameter list called a variable-length parameter list which is described next.

Since a standard 360 computer uses 24 byte storage addresses the left-most byte of an address constant is usually zero. In a variable length parameter list bit zero of the left-most byte of the last parameter

address constant is set to 1 to show that it is the last item in the list. The example above then would be written as:

```
PAR  DC   A(X)       address of X
     DC   A(Y)       address of Y
     DC   XL1'80'    turn on bit zero.
     DC   AL3(Z)     address of Z
```

if it generated a variable-length parameter list. As a matter of fact FORTRAN does generate variable-length parameter lists. Note though that programs expecting a fixed length parameter list will work with a variable-length parameter list, provided it is a least as long as the fixed-length list they are expecting.

REGISTER ASSIGNMENTS

Of the sixteen general purpose registers, five are assigned for use in the calling conventions. The use of the general registers differs slightly depending upon whether an R or S type call is being made.

The following table specifies exactly what each register is used for during a call:

Register Number	Contents
0	Parameter to be passed in R type sequences. Result to be passed back in R and S type sequences.
1	Parameter to be passed in R type sequences. Address of a parameter list in S type sequences.
2-12	Not used as a part of the calling sequence. Must be saved and restored by the called program. The save area is usually used for this.
13	The address of the save area provided by the calling program to be used by the called program.
14	Address of the location in the calling program to which control should be returned after execution of the called program.
15	Address of the entry point in the called program at the time of the call. A return code at the time of the return that indicates to the calling program whether or not an exceptional condition occurred during processing of the called program. The return code should be zero for a normal return or a multiple of four for various exceptional conditions.

General Purpose Register Linkage Conventions

Notice that it is the called program's responsibility to save and restore registers 2-12 in the save area provided by the calling program. There are two reasons for this. First of all only the called program knows how many of the registers from 2-12 it is going to use. Since a register need be saved and restored only if it is actually going to be changed, the called program may be able to save some time by saving and restoring only those registers which it will use. Secondly, the called program requires addressability over the area in which it will save registers upon entry, since any attempt to acquire the address of a save area would destroy some of the registers which are to be saved. Furthermore, the save area should not be a part of the called program since that would prevent it from being re-entrant (shareable). This means the calling program should provide the save area in which registers are saved and restored. And so we have the

called program saving and restoring registers 2-12 in a save area provided by the calling program.

The calling conventions are quite different with floating point registers. Since a large percentage of programs do not leave items in floating point registers across subroutine calls it seems rather wasteful to always save and restore the floating point registers. So the convention has been established that the calling program must save and restore those floating point registers which contain items which are wanted. Also, programs which return a single floating point result quite frequently do so via floating point register 0.

RETURNING RESULTS

There are in the OS Type I calling conventions four ways in which a subroutine can return a result. These are:

1. Value of result in general purpose register 0.
2. Value of result in general purpose register 1.
3. Value of result in floating point registers. (usually 0)
4. Value of a parameter from the parameter list changed.

The particular method used depends upon whether the R or S type convention is used and whether the called program can be used as a function in arithmetic statements.

The first three methods are used by R type calling conventions for all returned results. The contents of each of the registers depends upon the particular called program and are described in the subroutine writeup for each subroutine using the R type calling conventions.

The first, third, and fourth methods are used by S type calling conventions for all returned results. The first and third methods are used by function subprograms whose calls can be embedded in FORTRAN and MAD statements. The choice of general register 0 or floating point register 0 depends upon whether the result returned is integer or floating point mode, respectively. An example of subroutines which return results in this manner are the FORTRAN IV Library Subprograms, such as EXP, ALOG, or SIN. The fourth method can be used by a subprogram. An example would be a subprogram called by the statement

```
CALL MATADD(A,B,C,M,N)
```

which might add the MxN matrices A and B together and store the result in C.

MTS-130-0

12-1-67.

SAVE AREA FORMAT

The save area is an area belonging to the calling program which the called program uses to save and later restore general purpose registers. The address of the save area is passed to the called program by the calling program via general purpose register 13. The save area has a very rigid format and is described in the table:

Word	Displacement	Contents
1	0	Used by FORTRAN, PL/I, and other beasties for many devious purposes. Don't touch!
2	4	Address of the save area used by the calling program. Forms a backward chain of save areas stored by calling program.
3	8	Address of the save area provided by the called program for programs it calls. Forms a forward chain of save areas.
4	12	Return address. Contents of register 14 at time of call.
5	16	Entry point address. Contents of register 15 at time of call.
6	20	Register 0 contents.
7	24	Register 1 contents.
8	28	Register 2 contents.
9	32	Register 3 contents.
10	36	Register 4 contents.
11	40	Register 5 contents.
12	44	Register 6 contents.
13	48	Register 7 contents.
14	52	Register 8 contents.
15	56	Register 9 contents.
16	60	Register 10 contents.
17	64	Register 11 contents.

| 18 | 68 | Register 12 contents. |

SAVE AREA FORMAT

There are two things to be noted about the save area format, namely who sets what parts of the save area and how these areas might be set up. The calling program is responsible for setting up the second word of the save area. This is to contain the address of the save area which was provided when the calling program was itself called. Although this is technically set up by the calling program as a part of the call, most programs set up the save area they will provide to subroutines they call once and leave its address in general register 13. The work then does not need to be repeated for each call. The called program is responsible for setting up the third through eighteenth words of the save area. The called program usually saves the general registers which it will use as a part of its initialization procedure and restores the registers as a part of the return procedure. Notice that the save area format is amenable to use of the store multiple and load multiple instructions for saving and restoring blocks of registers. All of this will be made clearer in the examples at the end.

Some system subroutines (notably GETSPACE, FREESPAC, and a few others) do not require that a save area be provided for them. For these subroutines general register 13 need not be set up before a call and its contents are preserved by the called subroutine. The subroutines which need no save area are clearly marked as such in the MTS subroutine writeups. Notice that it is all right to provide a save area to one of these subroutine; it will simply be ignored.

CALLING PROGRAM RESPONSIBILITIES AND CONSIDERATIONS

The calling program is responsible for the following:

1. Loading register 13 with the address of the save area and setting up the second word of the save area.
2. Loading register 14 with the return address.
3. Loading register 15 with the entry point address.
4. Loading registers 0 and 1 with the parameters in an R type call or loading register 1 with the address of the parameter list in an S type call.
5. Saving floating point registers, if necessary.
6. Transferring to the entry point of the subroutine.
7. Restoring floating point registers, if necessary.
8. Testing the return code in register 15, if desired.

After the return from a subroutine, the status of the program will be as follows:

1. In general, the contents of the floating point registers will be unpredictable unless saved and restored by the calling program.
2. The contents of general registers 2 through 14 will be restored to

their contents at the time the called program was entered.
3. The program mask will be unchanged.
4. The contents of general registers 0, 1, and 15 may be changed.
5. The condition code may be changed.

Note that general registers 0 and 1 and floating point register 0 may contain results in the case of R type subroutine calls or a function subprogram. General register 15 will normally contain a return code, indicating whether or not an exceptional condition occurred during processing of the called program.

CALLED PROGRAM RESPONSIBILITIES AND CONSIDERATIONS

The called program is responsible for the following:

1. Saving the contents of general registers 2 through 12 and 14 in the save area provided by the calling program. These registers need be saved only if the called program modifies these registers.
2. Setting up the third word of the save area with the address of the save area which will be provided to subroutines it will call.
3. Restoring the contents of general registers 2 through 14 before returning to the calling program.
4. Restoring the program mask if changed.
5. Loading general registers 0 and 1 and floating point register 0 with the result in the case of R type subroutine calls or a function subprogram.
6. Loading general register 15 with the return code.
7. Transferring to the return location.

EXAMPLE CALLING SEQUENCES

This section will describe and give the assembly language statements for the typical machine instructions necessary to implement OS Type I calling conventions.

A typical entry point might consist of the following statements:

```
          USING SUBRA,12        12 will be a base register
SUBRA     STM   14,12,12(13)    save registers
          LR    12,15           set up 12 as the base register
          LA    11,SAVE         this is save area provided for others
          ST    11,8(0,13)      set up forward pointer
          ST    13,4(0,11)      set up backward pointer
          LR    13,11           set up for any calls we issue
          LR    11,1            get parameter pointer into non-volatile
                                reg.
          •
          •
```

```
          •
SAVE      DS    18F           save area we provide for others
```

Inside a subroutine that began with the entry sequence given above, the value of the second parameter in the parameter list could be put into general purpose register 3 with the following sequence:

```
          •
          •
          •
          L     3,4(0,11)     pick up second adcon from parameter list
          L     0,0(0,3)      pick up value of parameter
          •
          •
          •
```

Inside a subroutine that began with the entry sequence given above, another subroutine, SUBRB, could be called using the following sequence. Remember that register 13 already points to the correct save area:

```
          •
          •
          •
          LA    1,PARLIST     set up parameter list address
          L     15,=V(SUBRB)  set up entry point address
          BALR  14,15         set up return address and branch to the
                              subroutine
          B     *+4(15)       test return code via a transfer table
          B     AOK           RC=0
          B     BAD1          RC=4
          B     BAD2          RC=8
          •
          •
          •
AOK       •••                 normal return to here
          •
          •
          •
PARLIST   DC    A(PAR1)       first parameter address
          •
          •
          •
```

Finally, a subroutine that began with the entry sequence given above could return to the program that called it with the following sequence:

```
          LE    0,RESULT      floating point result to FPR 0.
          L     13,4(0,13)    use back pointer to get right save area.
          LM    14,12,12(13)  restore registers.
          SR    15,15         indicate a zero return code (no errors)
          BR    14            return to what called us
          •
```

•
•

It should be pointed out that although the above sequence are typical of the instructions used to implement the calling conventions, many variations are possible.

MACROS FOR CALLING SEQUENCES

There are two sets of macro definitions in the MTS Macro Library which can be used to help generate calling sequences. These are the OS macros SAVE, CALL, and RETURN; and the macros ENTER and EXIT. The more useful of these macros are ENTER, CALL, and EXIT. Besides these there is a set of macros which generate the entire calling sequences for many of the system subroutines and IOH/360. More details may be found in section MTS-255.

BATCH USER'S GUIDE

BATCH JOBS

MTS batch jobs are those which are turned in at THE WINDOW for an MTS receipt card and are run in the order that they are turned in. The output may then be picked up at THE WINDOW as soon as the job has been run. A batch job must not require any interaction.

ADVANTAGES AND DISADVANTAGES OF BATCH

Although many users will need to have interactive capability, many users will find that batch is satisfactory and perhaps even advantageous to them. There will probably be few if any interactive users who do not make at least occasional use of batch.

Batch has several advantages, even for those who have access to remote terminals. There is at present no way to use a high speed line printer or card reader or card punch directly from a terminal. But when running in batch mode one uses these devices at the Computing Center. Therefore, a user who does some of his work on a terminal may wish to create files containing the information he wishes to be printed and then either turn in a batch job to list these files or run *BATCH from his terminal to create a batch job to do the listing (see the writeup on *BATCH in section 280).

Batch may often be more economical than an interactive use of MTS. Since the charge for a job is based, in part, upon elapsed real-time, the terminal user will find that he is being charged something for just sitting and thinking. When running in batch, not only does the card reader not have to think about the next line to input but the input-output rates are much higher in batch. A disadvantage may arise in batch when the user wishes to compile and execute in the same job. Since there is presently no way for a run command to be made conditional, every run command will be executed even though the compilation which was to produce the object file was not successful. Usually a program interrupt occurs early in the execution of this file. This effect may or may not be desirable but the user should be aware of it.

DIFFERENCES BETWEEN BATCH AND TERMINAL USE

There are two commands which are not legal in batch ($SOURCE and $SINK) and three commands which are slightly different ($EMPTY, $DESTROY, and $RUN). $EMPTY and $DESTROY differ only in that no confirmation of the command is needed. $RUN differs in that *SOURCE* is set to the card reader containing the input stream, that *SINK* is the printer associated with this batch run and *PUNCH* is the punch associated with this batch run. *PUNCH* does not exist for terminal users. In batch, if the user does not specify global limits for time, pages, and cards on his $SIGNON card, default values will be supplied automatically and his job will be held to those limits. See the section Limits Specification, section MTS-225, for details.

USEFUL HINTS FOR RUNNING A BATCH JOB

1. It is often necessary to create an end-of-file while running a batch job. A $ENDFILE placed at the point where an end-of-file is needed will accomplish this. In addition there is an automatic end-of-file at the end of each batch job.

2. Any object deck in the source stream should be terminated with either an LDT card (see loader section) or a $ENDFILE.

3. It is strongly recommended that the user place his name enclosed by primes as the last parameter on the $SIGNON card.

4. Occasionally it is necessary to rerun a batch job and a user should take this possibility into account. For example, a user may create and give contents to a file in the first part of his job and empty and give new contents to the same file later. If it became necessary to rerun this job after it had already given the second contents to the file, it would not run the same if any of the line numbers in the second contents differed from the line numbers in the first. To solve this problem, instead of $CREATE use either $CREATE followed by $GET and $EMPTY or $DESTROY followed by $CREATE.

EXAMPLES OF MTS BATCH JOBS

1. Sample FORTRAN compilation and run.

```
$SIGNON  X007  'JAMES BOND'
$RUN *NEWFORT; 0=-OBJECT
     .
     .
     .
```

```
        FORTRAN program
        .
        .
        .
$ENDFILE
$RUN -OBJECT *SINK*; 1=*SOURCE* 2=*SINK*
        .
        .
        .
        data
        .
        .
        .
```

Comments: This job will compile the FORTRAN program in the source stream, putting the listing on the printer and the object program in a temporary file called -OBJECT. The object program is then run producing a map on the printer. The program will use logical unit 1 for input from the job stream and write the output on logical unit 2 which will be the printer.

2. Batch job to initialize the file PHROG from cards in the input stream.

```
$SIGNON  P314   'G.J.  NOHOPE'
$CREATE  PHROG
$EMPTY   PHROG
$GET PHROG
$NUMBER
        .
        .
        .
        cards to go into file
        .
        .
        .
$UNNUMBER
$SIGNOFF
```

Comments: The $EMPTY and $GET commands are is only important if the job is to be rerun. See above.

3. Batch job to list the file LISTING-B on the printer.

```
$SIGNON Q123  'FIFO STACK'
$LIST  LISTING-B
$SIGNOFF
```

4. Batch job to assemble a program, punching the object deck produced (default case).

```
$SIGNON  XXXX   'L.  USER'
$RUN *ASMBLR
        .
        .
```

```
      .
      assembly input
      .
      .
      .
$SIGNOFF
```

5. Batch job to run the object deck produced in the previous example. No map is wanted, and the program will read from SCARDS and print on SPRINT.

```
$SIGNON  XXXX  TIME=20  PAGES=62  'L.  USER'
$RUN
      .
      .
      .
      object deck
      .
      .
      .
$ENDFILE
      .
      .
      .
      data for the program
      .
      .
      .
```

TERMINAL USER'S GUIDES

This section contains usage information for the various types of console and terminal devices supported by MTS, such as teletypes, 2741s, etc.

MTS-151-0

12-1-67

TELETYPE USER'S GUIDE

(For Teletypes through 2702)

Introduction

This document gives instructions for initiating, operating, and terminating use of a teletype terminal calling in on numbers that go through a 2702 transmission control in MTS. (See section MTS-170 for a description of teletype use through the Data Concentrator – which is being used depends on the telephone number called.) The teletype terminal may be either a model 33 or 35, ASR or KSR model connected to the Bell Telephone System dialable network through an upright data phone and must be capable of half-duplex operation. A teletype with an inverted data phone or connected to the TWX network cannot be used as an MTS terminal. Although an ASR teletype with paper tape equipment may be used via the keyboard, MTS has no provisions for receiving or transmitting information via paper tape.

Teletypes are connected to the IBM System/360 through an IBM 2702 Telecommunications Control using a Telegraph Type II adapter. The IBM 2702 is connected to the telephone system through model 103A2 data sets. These data sets are assigned a block of telephone numbers in a trunk-hunting sequence within the University Centrex System. The telephone number is listed under 2702 - TELETYPE PORTS in MTS-270.

Initiation Procedure

To ready the teletype for use with MTS, proceed as follows:

1. If the teletype has the option to run in either half-duplex or full-duplex mode, place it in half-duplex mode. On some model 35 teletypes this is accomplished by twisting the HDX switch, which is located to the left of the keyboard, in a clockwise direction. On a model 33 and some model 35 teletypes flip the toggle switch located above the telephone dial to the HDX position.

2. Press the button labelled ORIG located beneath the telephone dial. You should now hear a dial-tone over a speaker inside the teletype. The volume control for the speaker is a knurled knob located to the right of the keyboard of a model 35 teletype or on the front panel of a model 33 teletype.

3. Dial the telephone number of the IBM 2702 - TELETYPE PORT. The

telephone number is listed in MTS-270. The first telephone number listed will search all of the numbers and connect you to the first available one. If your teletype is in the University Centrex System, you need dial only the last five digits of the telephone number.

4. If MTS is on the air and there is a free line, your call will be answered. Either a busy signal or a ring with no answer indicates that MTS is not on the air or there are no free lines.

5. After answering your call MTS types out:

   UNIVERSITY OF MICHIGAN TERMINAL SYSTEM: ANN ARBOR,MICHIGAN.
   WHO ARE YOU?

   If your teletype does not respond with an automatic answerback, you must type one. To do this, press the K button to the left of the keyboard and type:

   a. RETURN
   b. LINE FEED
   c. A sequence of up to ten characters.
   d. RETURN
   e. LINE FEED
   f. X-OFF

6. If you have not already done so, press the K button to the left of the keyboard. This connects the keyboard to the telephone line.

7. When MTS types the character #, it is operational and ready for a command line. The first command you issue should be $SIGNON. For a description of the commands see MTS-230.

Keyboard Operation

During normal operation, the teletype is in one of three modes: receive mode, transmit mode, or idle mode. Since the teletype keyboard does not lock or otherwise indicate the current mode, it is the user's responsibility to be aware of the current mode and act accordingly. A description of each mode and the action to be taken by the user in that mode follows:

1. The teletype is in receive mode whenever a message is being typed on the teletype by MTS or some program running under MTS. The first character of the output line is a prefix character which indicates "who is speaking". These prefix characters are defined in MTS-220. The terminal user should not type on the keyboard while the teletype is in receive mode.

2. The teletype is in transmit mode whenever MTS or some program running under MTS is waiting for the terminal user to type in a line

at the teletype. A prefix character is typed out as the only character of the line to show that input is expected and indicate "who has requested it". These prefix characters are defined in MTS-220.

The terminal user should respond by typing in the requested line, terminating it by typing the character X-ON (control Q) or the character X-OFF (control S). Note that the RETURN character is treated just as any other character and will not terminate a line. There are two classes of characters which have special significance. The first class are characters which can be used to edit the line as it is typed in before it is given to the routine requesting it. The characters are described briefly in MTS-220 and in more detail in the table below. Characters of this class will be treated as ordinary characters if preceded by the "literal next character" character. The second class of characters are those which are treated peculiarly by the IBM 2702. These characters cannot be treated as ordinary characters because of hardware limitations.

If a transmission error is detected while the teletype is in transmit mode, the prefix character is retyped on the same line and all characters typed up to this point are ignored. The teletype remains in transmit mode waiting for the input line.

An input line may consist of up to 96 characters, not including the line termination character. If this count is exceeded, an attention is assumed. See the next section of the writeup for details about attentions.

3. The teletype is in idle mode when it is neither in receive mode nor in transmit mode. That is, the teletype is in this mode when the program or command being performed is neither writing information on the teletype nor waiting for a input line from the teletype. For instance, the teletype will be in idle mode if the program the user is running is computing with no input or output, or is in an infinite loop, and so forth. While in idle mode, the teletype will "grunt" for about a second once each thirty seconds. This is meant to reassure the user that MTS has not forgotten him. The terminal user should not type on the keyboard while the teletype is in idle mode.

CHARACTER	FUNCTION
CONTROL A (SOM)	The previous character is deleted.
CONTROL C (EOM)	A logical end-of-file return is presented to the program. Any other contents of the line are not returned.
CONTROL N (SO)	The current input line is deleted. The teletype returns to transmit mode for the line to be retyped.
CONTROL Z (S2)	The next character typed is treated as an ordinary character, even if it is a CONTROL A, CONTROL C, CONTROL N or a CONTROL Z. This has no effect on characters below.
RUBOUT	This character is completely ignored by the IBM 2702.
CONTROL Q (X-ON)	This character terminates a transmit operation in a normal manner.
CONTROL S (X-OFF)	This character terminates a transmit operation in a normal manner.
CONTROL E (WRU)	This character terminates a transmit operation in a normal manner, unless it is the only character typed, in which case it causes MTS to give its answer-back and the transmit to be retried.
CONTROL D (EOT)	This character terminates a transmit operation by hanging up the teletype. This is not the approved way to terminate a session.

Table I: Characters which have special significance during transmit operations. Control characters are typed by holding down the control key while typing the character, control-shift characters by holding down both the control and the shift keys while typing the character

Attentions

An attention is a signal to MTS to interrupt whatever it is doing for you and return for further command lines. An attention can be used, for instance, to interrupt a $LIST command after you have seen all which you wish to see from the file, to interrupt a program which is executing to check its progress and so forth. A program that has been interrupted via an attention can be continued at the point of the most recent interrupt by using the $RESTART command. There are a few times when an attention will not interrupt MTS. These are: during the initial signon procedure, during the signoff procedure, and if you have just issued an attention which has not taken effect yet.

To issue an attention to MTS press the BREAK button. This button is located to the left of the keyboard on a model 35 teletype and at the right end of the keyboard on a model 33 teletype. After pressing the BREAK button, press the BREAK RELEASE button located above the telephone dial and the K button located to the left of the keyboard. MTS should now type out the message:

ATTENTION ASSUMED.

The exact action taken next depends on what mode the teletype was in at the instant the the BREAK button was pushed:

1. If the teletype is in receive mode, the exact action taken depends on what MTS was doing at the moment of the attention. Three situations are possible:

   a. If MTS has already set up the next line of output and was waiting for the current line to finish so that it could type this new line out, the new line is typed out before the interrupt is taken.

   b. If MTS was waiting for the current line to finish so that it could request a new line from the user, the new line is requested before the interrupt is taken. This is necessary in case the user decides to continue after the interrupt.

   c. If MTS was not waiting for an input or output line from the teletype the interrupt is taken immediately.

2. If the teletype is in transmit mode, the line which was requested must be entered before the interrupt is taken. This is necessary in case the user decides to continue after the interrupt.

3. If the teletype is in idle mode, the interrupt is taken immediately.

When the interrupt is finally taken as described above, the message

ATTENTION INTERRUPT AT XXXXXXXX

is typed, where XXXXXXXX is the right half of the PSW at the time of the interrupt. The teletype is then placed in transmit mode waiting for a command line to MTS.

Normal Termination Procedure

If the teletype is not in transmit mode waiting for a command line, issue an attention. When the teletype does enter transmit mode waiting for a command line, type in a $SIGNOFF command. MTS will now close all your files, type out several lines of statistics for this session, and turn off the teletype.

Sample Session

Following is a sample session using MTS from a teletype. Because the edit and control characters are non-printing, a control character is indicated below by a lower-case letter, the letter being the one pushed with the control key down. (e.g., X-OFF is CONTROL-S which is printed as "s") At this session, a file was created and a FORTRAN program typed into it, the file was edited and then compiled, and finally the object deck was run. Messages to the right in parenthesis are editorial comments. Underlined characters were typed by the user.

```
UNIVERSITY OF MICHIGAN TERMINAL SYSTEM : ANN ARBOR,MICHIGAN.
WHO ARE YOU?e
UM CMPC B AA            (Answerback code from teletype)

#$SIGNON QQQs
#**LAST SIGNON WAS:  12:15:42   05-01-67
#  USER "QQQ." SIGNED ON AT  13:31.50 ON 05-01-67
#$CREATE DEMOSs
#  FILE "DEMOS         " HAS BEEN CREATED.
#$NUMBERs
#      1100 FORMAT(A4)s
#      2READ (5,100) ALPHAs
#      3WRITE (6,100) ALPHAs
#      4GO TO 1s
#      5ENDs
#      6$UNNUMBERs
#1.5 1  CONTINUEs
#$LIST DEMOSs
>       1      100 FORMAT(A4)
>       1.5     1  CONTINUE
>       2      READ (5,100) ALPHA
>       3      WRITE (6,100) ALPHA
>       4      GO TO 1
>       5      END
#END OF FILE
#$RUN *FORTEDIT;SCARDS=DEMOS SPUNCH=-SOURCEs
#EXECUTION BEGINS
#EXECUTION TERMINATED
#$LIST -SOURCEs
>       1      100    FORMAT(A4)
>       2       1     CONTINUE
>       3             READ (5,100) ALPHA
>       4             WRITE (6,100) ALPHA
>       5             GO TO 1
>       6             END
#END OF FILE
```

```
MTS-151-0

12-1-67


#$RUN *NEWFORT;SCARDS=-SOURCE 0=-OBJECT PAR=NOMAPs
#EXECUTION BEGINS

FORTRAN IV G LEVEL 0, MODE 0            MAIN                DATE = 13:3


 0001         100    FORMAT(A4)
 0002          1     CONTINUE
 0003                READ (5,100) ALPHA
 0004                WRITE (6,100) ALPHA
 0005                GO TO 1
 0006                END


FORTRAN IV G LEVEL 0, MOD 0             MAIN                DATE = 13:3

      TOTAL MEMORY REQUIREMENTS 000132 BYTES
#EXECUTION TERMINATED
#$RUN -OBJECT -MAP ; 6=*SINK*s
#EXECUTION BEGINS
  5     WAS CALLED BUT NOT SPECIFIED. ENTER UNIT NAME OR "CANCEL"
?*SOURCE*s
 ABCDs
 ABCD                              (Program types back its input)
 ABCDnsDELETED                     (An example of deleting a line)
 ABCaYZs                           (Example of deleting a character)
 ABYZ
 X                                 (The break key was pressed here)
ATTENTION ASSUMED.
 ABBAs                             (Line requested again in case of
 ATTENTION INTERRUPT AT 4E0396AA                                restart)
#$RESTART
 ABBA                              (Note interrupt was transparent)
 csENDFILE                         (End-of-file generated)


  IHC217I

#EXECUTION TERMINATED
#$SIGNOFF
#**** OFF AT 13:41.14
#**** ELAPSED TIME       564.593   SEC.
#**** CPU TIME USED        9.5     SEC.
#**** STORAGE USED      5051.616   PAGE-SEC.
#**** FILE STORAGE      1234       PAGE-MIN.
```

Translation to and from ASCII

The IBM/360 computer represents characters internally in a code referred to as EBCDIC which stand for Extended Binary Coded Decimal Interchange Code. This code represents each character uniquely in eight bits, with a hidden ninth bit for parity. Hence there are 256 possible characters within the encoding. Of these 256 possible characters, approximately 100 have been assigned commonly used graphics and meanings. The others are more or less available for arbitrary use.

Model 33 and 35 teletypes, on the other hand, encode data in a code referred to as ASCII-8. This code represents each character in an eight bit code, where seven of the bits are unique for each character and the eighth bit is optionally used for parity checking. Hence there are two representations for each character, an even-parity representation and an odd-parity representation. There are exactly 128 characters in ASCII-8 and each is assigned either a graphic or some control function.

The teletype support routines in MTS then must translate every input character from a teletype into the equivalent EBCDIC character and must translate every character going to the teletype into ASCII-8. In order to accomplish this, a mapping function was set up. The criteria used in this function are:

1. Every unique ASCII-8 character typed in at the teletype is assigned a unique representation within the IBM/360.

2. Input characters are not checked for correct parity. This means the teletype can generate even parity, odd parity, or arbitrary parity.

3. All characters sent back to the teletype are sent in their even-parity representation.

4. Any EBCDIC character which does not correspond to an ASCII-8 character (there are 256-128=128 of these) will be converted to the ASCII-8 NULL character if sent as output to a teletype.

5. Every ASCII-8 character is translated into an EBCDIC character having the same graphic or control function if such exists. If there is no EBCDIC character with the same meaning, one of the left-over EBCDIC characters was chosen rather arbitrarily.

Following is a table showing the correspondence between the ASCII-8 character and EBCDIC representation. The X in the eighth bit of the ASCII-8 character can be either 0 or 1 on input and is set to force even parity on output.

EBCDIC ENCODING (HEXADECIMAL)	ASCII-8 ENCODING (BITS 1...8)	NAME	TTY KEY	FUNCTION
00	0000 000X	NULL	C-S-P	BLANK TAPE
04	0010 100X	-TAPE	C- T	TAPE PUNCH OFF
05	1001 000X	HTAB	C- I	HORIZONTAL TAB
07	0000 100X	DL	C- P	
12	0111 111X	ESC		ESCAPE KEY
15	1011 000X	RETURN		CARRIAGE RETURN
17	1111 111X	RUBOUT		ALL HOLES PUNCH
18	1111 100X	S7	C-S-O	
19	0111 100X	S6	C-S-N	
1A	1011 100X	S5	C-S-M	
1B	0011 100X	S4	C-S-L	
1C	1101 100X	S3	C-S-K	
1D	0101 100X	S2	C- Z	
1E	1001 100X	S2	C- Y	
1F	0001 100X	S0	C- X	
21	1000 000X	SOM	C- A	START OF MESSAGE
22	1011 111X	ALT MD		ALTERNATE MODE
25	0101 000X	LF		LINE FEED
26	1100 000X	EOM	C- C	END OF MESSAGE
2B	0011 111X	CN FM		SPECIAL FORM
2C	1110 100X	LEM	C- W	
2D	0110 100X	SYN	C- V	
2E	1010 100X	ERROR	C- U	
2F	1000 100X	X-ON	C- Q	TAPE READER ON
30	1111 000X	SI	C- O	
31	0111 000X	SO	C- N	
32	0011 000X	FORM	C- L	FORM FEED
33	1101 000X	VT	C- K	VERTICAL TAB
34	0100 100X	TAPE	C- R	TAPE PUNCH ON
35	1100 100X	X-OFF	C- S	TAPE READER OFF
37	0010 000X	EOT	C- D	END OF TRANSMN
3B	0001 000X	FEO	C- H	
3C	1110 000X	BELL	C- G	RINGS BELL
3D	0110 000X	RU	C- F	ARE YOU?
3E	1010 000X	WRU	C- E	WHO ARE YOU?
3F	0100 000X	EOA	C- B	END OF ADDRESS
40	0000 010X	SPACE		BLANK SPACE
4B	0111 010X	.		PERIOD (DECIMAL)
4C	0011 110X	<		LESS THAN
4D	0001 010X	(		LEFT PAREN
4E	1101 010X	+		PLUS OR ADDITION
50	0110 010X	&		AMPERSAND
5A	1000 010X	!		EXCLAMATION PT
5B	0010 010X	$		DOLLAR SIGN
5C	0101 010X	*		ASTERISK
5D	1001 010X	)		LEFT PAREN
5E	1101 110X	;		SEMI-COLON

60	1011 010X	-		NEG. OR MINUS
61	1111 010X	/		ORDINARY SLASH
6B	0011 010X	,		COMMA
6C	1010 010X	%		PERCENT
6E	0111 110X	>		GREATER THAN
6F	1111 110X	?		QUESTION MARK
7A	0101 110X	:		COLON
7B	1100 010X	#		POUND SIGN
7C	0000 001X	@		AT SIGN
7D	1110 010X	'		APOSTROPHE
7E	1011 110X	=		EQUAL SIGN
7F	0100 010X	"		DOUBLE QUOTE
81	1000 011X	a		LOWER CASE A
82	0100 011X	b		LOWER CASE B
83	1100 011X	c		LOWER CASE C
84	0010 011X	d		LOWER CASE D
85	1010 011X	e		LOWER CASE E
86	0110 011X	f		LOWER CASE F
87	1110 011X	g		LOWER CASE G
88	0001 011X	h		LOWER CASE H
89	1001 011X	i		LOWER CASE I
91	0101 011X	j		LOWER CASE J
92	1101 011X	k		LOWER CASE K
93	0011 011X	l		LOWER CASE L
94	1011 011X	m		LOWER CASE M
95	0111 011X	n		LOWER CASE N
96	1111 011X	o		LOWER CASE O
97	0000 111X	p		LOWER CASE P
98	1000 111X	q		LOWER CASE Q
99	0100 111X	r		LOWER CASE R
A2	1100 111X	s		LOWER CASE S
A3	0010 111X	t		LOWER CASE T
A4	1010 111X	u		LOWER CASE Q
A5	0110 111X	v		LOWER CASE V
A6	1110 111X	w		LOWER CASE W
A7	0001 111X	x		LOWER CASE X
A8	1001 111X	y		LOWER CASE Y
A9	0101 111X	z		LOWER CASE Z
AD	1101 101X	[	S-K	LEFT BRACKET
BA	0011 101X	\	S-L	BACKWARD SLASH
BB	0111 101X	↑		UPWARD ARROW
BC	1111 101X	←		LEFTWARD ARROW
BD	1011 101X	]	S-M	RIGHT BRACKET
C1	1000 001X	A		UPPER CASE A
C2	0100 001X	B		UPPER CASE B
C3	1100 001X	C		UPPER CASE C
C4	0010 001X	D		UPPER CASE D
C5	1010 001X	E		UPPER CASE E
C6	0110 001X	F		UPPER CASE F
C7	1110 001X	G		UPPER CASE G
C8	0001 001X	H		UPPER CASE H
C9	1001 001X	I		UPPER CASE I
D1	0101 001X	J		UPPER CASE J

D2	1101 001X	K	UPPER CASE K
D3	0011 001X	L	UPPER CASE L
D4	1011 001X	M	UPPER CASE M
D5	0111 001X	N	UPPER CASE N
D6	1111 001X	O	UPPER CASE O
D7	0000 101X	P	UPPER CASE P
D8	1000 101X	Q	UPPER CASE Q
D9	0100 101X	R	UPPER CASE R
E2	1100 101X	S	UPPER CASE S
E3	0010 101X	T	UPPER CASE T
E4	1010 101X	U	UPPER CASE U
E5	0110 101X	V	UPPER CASE V
E6	1110 101X	W	UPPER CASE W
E7	0001 101X	X	UPPER CASE X
E8	1001 101X	Y	UPPER CASE Y
E9	0101 101X	Z	UPPER CASE Z
F0	0000 110X	0	NUMERAL 0
F1	1000 110X	1	NUMERAL 1
F2	0100 110X	2	NUMERAL 2
F3	1100 110X	3	NUMERAL 3
F4	0010 110X	4	NUMERAL 4
F5	1010 110X	5	NUMERAL 5
F6	0110 110X	6	NUMERAL 6
F7	1110 110X	7	NUMERAL 7
F8	0001 110X	8	NUMERAL 8
F9	1001 110X	9	NUMERAL 9
FE	1101 111X		
FF	0000 011X		

MTS-152-0

12-1-67

IBM TERMINAL TYPE 2741 USER'S GUIDE

(For 2741s through the 2702)

Introduction

This guide gives instructions for the use of the 2741 as a remote MTS terminal when connected through the 2702 (see section MTS-170 for description of use through the Data Concentrator - which is being used depends on the telephone number called). The transmission of information between the 2741 and the CPU is accomplished via the 2702 Transmission Control and 2870 Multiplexor Channel. Common carrier connections are utilized and require the use of a standard data phone and a 103A data set. The telephone numbers for 2741 terminals are different than those for teletypes and are listed in the Data Set Directory MTS-270-1.

The 2741 may be used as an ordinary electric typewriter when it is not being used as a communications terminal. For this type of use the main power switch (keyboard) should be "ON" and the "LOCAL-COMMUNICATIONS" switch (left side panel) should be set to LOCAL.

Terminal Procedures:

1. Initiation

   A. For use as an MTS terminal the LOCAL-COMM switch must be set to "COMM" and the main power switch must be "ON" before dialing the computer telephone number. The position of the "golf ball" carrier is not important; it will be reset automatically.

   B. To make the telephone connection depress the "TALK" button on the data phone panel, pick up the hand set, and dial the proper number. When the high pitched tone is heard depress the "DATA" button and hang up the hand set. Either a busy signal or a ring with no answer indicates that MTS is not available at that time or that there are no free lines.

   C. A completed call to the system will cause the message UNIVERSITY OF MICHIGAN COMPUTING CENTER, ANN ARBOR to be typed out. This is followed by a carrier return (with line feed) and then the typing of the character #. This indicates that the first

command line ($SIGNON xxxx) can be entered. After entering the "signon" command character string, the carrier return key should be depressed to indicate the end of the message.

2. Conversation Operation

A. Terminal modes: During operation the 2741 may be in one of three modes: receive, control-receive, or transmit. The keyboard (with the exception of the "attention" key) is locked except when in transmit mode. Normally the 2741 is placed in transmit mode only when MTS expects a line to be entered.

B. Entering MTS lines: Alphabetic characters in command lines are always converted to upper case before the command line is analyzed; thus $SIGNON and $signon produce the same effect. Data lines, however, are entered exactly as they are typed at the terminal. If automatic upper case conversion is desired for keyboard entry of alphabetic characters, one may use the $SET command to specify forced upper case conversion for lines read by the MTS monitor, and the @UC modifier in all other cases.

A line may be deleted by depressing the "attention" key. In addition to the line delete function three characters are assigned special control functions for the 2741 communications with MTS. These are:

1. Back space - this causes the preceding character of an input line to be deleted. Consecutive backspaces may be used to delete several previous characters or even an entire line; however, if an entire line is wiped out with backspaces and then the carrier return key is depressed a zero length line is transmitted to the MTS routines. (Note that his differs from a line delete via the attention key. A line deleted via "attention" is never transmitted to the MTS routines.

2. Cent sign - this is used to indicate logical end of file; the contents of the input containing a ¢ are not transmitted to MTS, only the end-of-file signal is transmitted.

3. Exclamation point - this is used as the "literal next character" character. Should it be desirable to actually enter a backspace, cent sign, or exclamation point into a command or data line, these characters can be preceded by one "exclamation point" . In this context the pair of characters is taken as a single character with the normal graphic value of the second rather than as a sequence of control characters.

The order for analyzing input lines is as follows.

a. A line is deleted if the attention key is depressed regardless of the contents of the input line.

b. Literal next characters are applied (note that literal next characters have no meaning unless they precede one of the three special characters and are ignored if out of context).

c. If any backspaces remain they are applied to delete the appropriate previous characters.

d. If an end-of-file character remains, a logical end-of-file is returned to MTS; otherwise the edited line is returned.

Any length of time may be used to enter a single input line via the 2741; however if there is no activity for a span of approximately 15 minutes the user and terminal will be automatically signed off. Actually a "timeout" occurs (in the 2702) if no character is entered within 28 seconds of the previous character. In this event all characters transmitted are saved and the 2702 is again prepared to receive text from the 2741 so that another segment of the input line can be entered. An input line is thus accumulated over a relatively long time interval. It may occur that a user enters a character while the 2702 is being reset for the next line segment (very unlikely but it can happen); in this case the message LINE DELETED: LOST DATA will appear and the entire line will have to be reentered. Input lines may contain up to 128 characters.

C. Attention interrupts: An attention interrupt is a signal to MTS to interrupt whatever it is doing for you and to return for another command line. One may interrupt the execution of a program, the listing of a file, etc. by depressing the attention key. If an attention interrupt is desired while the 2741 is in transmit mode the attention key should be depressed twice; the first time to cause line deletion and the second time during the typing of the "LINE DELETED" message. That is, attention interrupts are possible only when the 2741 is in receive mode, i.e. only when it is receiving a line from MTS or during execution of some program. The 2741 will type out the message ATTENTION INTERRUPT AT XXXXXXXX and return to transmit mode after typing #. At this time you may enter a new command including $RESTART which cause execution to resume where it was interrupted.

3. Termination Procedure: With the 2741 in transmit mode enter the command $SIGNOFF. After this command line is scanned MTS will properly close all of your files (this may take a few seconds) and then type out a summary of your session at the terminal. The line will be automatically disconnected.

MTS-153-0

12-1-67

1050 USER'S GUIDE

It is possible to use an IBM 1050 terminal in MTS by dialing the same numbers as are used for an IBM 2741. The 1050 must have several special features and RPQ's, primarily send and receive break. A complete User's Guide is in preparation and will be released as soon as possible.

2250 MODEL I DISPLAY USER'S GUIDE

This writeup gives instructions for the use of the 2250 Model I Display as a terminal running under MTS. Terminal and system interaction is strictly in character mode using only the alphanumeric keyboard and the programmed function keyboard. Various library files and user programs that run via a RUN command may (and do) use the light pen and graphics features of the 2250.

Initiation

When the 2250 is activated and ready for users, the screen is blank and the programmed function keyboard lights are lit up in a block "M". Pressing any programmed function keyboard button causes the standard reply "UNIVERSITY OF MICHIGAN TERMINAL SYSTEM" to be displayed on the screen and a cursor is positioned at the bottom of the screen. The $SIGNON command may now be entered via the alphanumeric keyboard.

Conversational Operation

In command mode and during RUN's that are not graphics programs, the last 20 input or output lines are displayed on the upper half of the screen, and space for two lines (148 characters) of characters to be input is provided at the bottom of the screen, with a cursor when the program (MTS or user program) requests input. These two lines form one logical line of up to 148 characters to be sent to the computer when requested.

Alphanumeric Keyboard. Striking a character key on the keyboard (when a cursor is displayed) causes that character to be input to the input region which is displayed at the bottom of the screen. END (hitting the 5 key while holding down the ALTN CODING key) causes the contents of that input region, up to the cursor, to be input to the program. CANCEL (hitting the 0 key while holding down the ALTN CODING key) causes the contents of the input region to be erased; nothing is sent to the program. The ADVANCE and BACKSPACE keys position the cursor along the input line. The JUMP key moves the cursor back to the first character in the input line. The space bar is not the same as ADVANCE: it enters a blank into the input line as well as advancing the cursor by one position. The CONTINUE key, if held down when a character key is hit, causes that character to be repeated across the input line until both keys are released.

Programmed Function Keyboard. All keys except 28 and 29 are legal only when they are lit.

Key 0 causes an attention interrupt to be generated.

Key 3 causes the display to revert to the last 20 lines. (way to get back to original position after scrolling)

Keys 11,16,18,23.
At any given time, up to the last 2000 lines of input/output are saved in virtual storage. (If this limit is exceeded, all but the last 4096 characters are thrown away) These lines can be up to 148 characters (input) or 255 characters (output) long. The display screen may be thought of as a "window" 74 characters wide and 20 lines deep, looking out over this region. In order to see everything, it is necessary to be able to scroll (move back and forth) the background up and down, and right and left behind this window. These four keys (11, 16, 18, 23) are arranged in a diamond, and pressing the obvious key starts scrolling in that direction. Pressing the key a second time stops it. Key 28 (see description) controls speed.

Key 15 releases all previous input/output lines saved except the last 4096 bytes.

Key 27 when pressed causes a "new output page" to be displayed. The last 20 lines are saved and the 20 line display area is blanked.

Key 28 is a slow-fast key which controls the rate at which output is displayed.
UNLIT: fast - as fast as can be received.
LIT: slow - about one line per second.
Each press of the key flips it between the fast and slow states. It is initially unlit.

Key 29 is a stop-go key which controls the accepting of output lines from the program.
UNLIT: go - lines are accepted and displayed at the speed specified by key 28.
LIT: stop - accept no more output lines for the time being.
Each press of the key flips it between the stop and go states. It is initially unlit.

Key 31 gives an end-of-file.

MTS-160-0

12-1-67

TAPE USERS GUIDE

INTRODUCTION

This document gives instructions for mounting and accessing data stored on magnetic tapes. Nine track tapes and seven track tapes written at either 200, 556, or 800 BPI with odd or even parity are currently supported.

BASIC CONCEPTS

In order to absolve any ambiguities which might arise the following definitions are offered.

7 and 9 Track Tapes

A 7 track tape frame contains 6 bits of data and one parity bit; a 9 track tape frame contains 8 bits of data and one parity bit. This property of the tape is a direct function of the tape unit upon which the tape was written; that is, all 7 track tapes must be read and/or written on 7 track tape units, all 9 track tapes must be read and/or written on 9 track tape units. All 7090 compatible tapes are 7 track tapes.

Odd and Even Parity

The above mentioned parity bit may be manipulated so that there are either an odd (odd parity) or an even (even parity) number of bits set to one in each tape frame. If binary records are to be written odd parity must be used to prevent loss of data. Common usage also requires that 7 track BCD tapes be written with even parity.

Densities

It is possible to write 7 track tapes in any one of three densities:

(a) 200 BPI which corresponds to our 7090 low density
(b) 556 BPI which corresponds to our 7090 high density
(c) 800 BPI which cannot be read by our 7090.

All 9 track tape drives write at 800 BPI density.

Translator and Data Convertor

Since 7 track tape characters contain only 6 bits of data and 360 data bytes contain 8 bits of data it seems like there is a problem getting from one to the other. To solve these problems the translator feature and data convertor features are provided. By playing the appropriate games it is possible to write a 7 track tape with either translator on, data convertor on, or both features off.

If translator feature is on an automatic conversion is invoked which translates most 8 bit EBDIC characters into their corresponding 6 bit BCD counterparts.

If data convertor feature is on 4 tape characters will be produced for each 3 data bytes. All 8 bits in each byte are thus transmitted. Data convertor on forces odd parity.

If neither translator or data convertor is specified the low order 6 bits of each data byte are transmitted to/from the tape. To insure accurate data transmission odd parity should be specified when both translator and data convertor are off.

9 track tape users should just forget all that noise. All 9 track tapes are written in odd parity at 800 BPI.

File Protect Ring

Since writing on a tape destroys any information which may have been stored upon it a file protection device is provided to prevent accidental erasure of save tapes. This device, hereafter called file protect ring, or just ring for short, must be inserted in a groove in the back of the tape reel to allow writing on the tape. The user must specify at the time the tape is mounted whether the ring is to be in (allowing writing) or out (prohibiting writing).

Record Size

Every normal read or write operation transmits a block of data to/from a tape unit. This block of data is called a record. Usually a record corresponds to one line in a file; but it is also possible to write very large records. One of the attributes associated with a tape is an upper bound to the size of a record.

End of Tape Area

Approximately 20 feet from the end of each reel of tape there is a silver strip known as the end of tape marker. There is enough tape after this marker to allow users who desire to write a short (5 records or less) trailer label. Users will not be allowed to write more than 5 records in this area.

Pseudo-Device Names

Because there will usually be more than one active user at any given time it is not always possible to predict with any great probability of success what particular tape unit will be free. Therefore all references to tapes should be made through pseudo-device names (PDN's). MTS will then determine which unit the PDN refers to and the appropriate read/write function will be performed. A PDN may be used exactly as if it were a "FDname".

A PDN consists of an asterisk followed by a string of from one to fourteen alphanumeric characters and terminated by another asterisk. For example, *NAME* and *TAPE* are valid pseudo-device names. Note that MTS preempts certain PDN's (see MTS-210). Those names may not be used as PDN's.

USING TAPES

Mounting a Tape

It is possible to have a tape mounted while in execution or command mode. In either case the file *MOUNT should be used - see MTS-280/44645, MTS-255/44645 and MTS-253/44645. The user should specify at the time of the mounting of the tape both the mode and record size of the volume if the default parameters are not to be used. A table of all legal modesets may be found later in this section. The current default mode is 5EN for 7 track tapes (corresponding to 7090 high density BCD tapes). Users wishing to write 7090 binary tapes should specify MODE=50F. 9 track tapes do not have a mode associated with them.

The default record size is 256 characters but users reading and/or writing large records may specify any maximum up to 32767 characters.

Removing Tapes

There are three ways to have a tape removed from a tape drive:
(1) Mount another tape on the same pseudo-device - that is, invoke *MOUNT using a PDN which has been used before for another tape. The first tape will be removed and the second mounted in its place. The mode and record size of the first tape will be used for the second tape unless explicitly reset.
(2) Invoke *DISMOUNT (see MTS-280/24627 and MTS-253/24627)
(3) Sign off MTS.

In either case if the last operation performed on the tape was a write, five end-of-file marks will be written on the tape to terminate it before it is unloaded.

Data Transmission

To transmit data to/from a tape in a program attach the PDN associated with the tape to any logical device. Then transmit to/from that logical device as if it were a sequential file. For example,

```
SPRINT=*TAPE*
```

In commands, use the PDN for a FDname. For example,

```
$COPY FILE1 TO *TAPE*
```

RETURN CODES

The return codes resulting from data transmission to/from tapes extend beyond those returned from other devices. They are

0	Normal return
4	(from read operations) End-of-file mark detected. (from write operations) End of tape strip sensed.
8	Tape load point has been sensed on backspace command.
12	User attempted to write more than five records in end of tape area.
16	Permanent read/write error (on read, tape will be positioned past bad record).
20	Attempt to write on a file protected tape.
24	Equipment malfunction.

Unless the device error exit has been set by the user, all return codes greater than 4 will be intercepted by MTS and the current RUN will be terminated. Control will be returned to the user in command mode. If the device error exit has been set the user may recover from the error in any way he chooses and still remain in execution. (see a description of the SETIOERR subroutine)

Control Functions

Users may want to initiate such control functions as rewinding and backspacing tapes. This is done by attaching the logical carriage control attribute to the tape. In this case the first three bytes of the "region" specified in the user's call are used to specify the control operation desired. A complete table of the tape controls is given below.

This same procedure allows users to change modes during execution. (perhaps to read a record with bad parity) The legal modesets are also listed below. In addition to changing the mode, the data is also transmitted on recognized modesets. In the case of a write the remaining "length"-3 bytes are written; for read, the record is read into the region beginning at the first byte of the region (and wiping out the modeset).

If logical carriage control was specified but the first 3 bytes are neither a legal control function nor legal modeset, the entire record is written (or read) as if logical carriage control had not been specified.

Tape Control Functions

Logical Command	Action Taken	Non-Zero Return Code Meaning
WTM	End of file record written.	none
WEF		
FSF	Tape spaced forward past next end of file record.	4=in end of tape area
FSR	Tape spaced forward one record.	4=tape mark sensed
BSF	Tape spaced backward past next end of file record	8=loadpoint sensed
BSR	Tape spaced backward one record.	4=tape mark sensed 8=loadpoint sensed
REW	Tape rewound to loadpoint.	none
RUN	Tape rewound and unloaded.	none
SRL	Resets maximum record length. (the length should be the five characters following the "SRL", left-justified with trailing blanks)	4=illegal length

Mode Setting Commands
(for 7 track tapes only)

(1) Type 1 are of the form "dCV"
(2) Type 2 are of the form "dpt"

where "d" is the code for density:
2 for 200 BPI
5 for 556 BPI
8 for 800 BPI
and "p" is the code for parity:
0 for odd
E for even
and "t" specifies whether or not the translator is on:
F for translator off
N for translator on

Since 7 track tapes can carry only 6 bits of information per tape character and 360 bytes have 8 bits per word a compromise was necessary. Type 1 records, written with DATA CONVERTOR ON write 4 tape characters for each group of 3 bytes, overlapping from one tape character to the next. Type 2 records maintain a one-to-one correspondence between core bytes and tape characters, but use only the low order six bits in each byte.

PAPER TAPE USER'S GUIDE

The Computing Center has installed on the 360/67 a 2671 paper tape reader (see IBM manual A24-3388) and a 1012 paper tape punch (see IBM manual A26-5776 plus RPQ 834427 description "1012 Attachment to System 360"). Programming support for these is underway, and a full writeup will be issued shortly. The routines are being constructed so that using paper tape will be like using magnetic tape, and the MOUNT and DISMOUNT subroutines will be used to mount and remove paper tapes.

DATA CONCENTRATOR USER'S GUIDE (TELETYPES, 2741S, REMOTE COMPUTERS)

THE DATA CONCENTRATOR

Terminal equipment with unusual characteristics can be attached to MTS using the facilities of the Data Concentrator, a special-purpose terminal control attached directly to the Model 67. The Data Concentrator consists of a Digital Equipment Corporation PDP-8 computer to which special interface equipment has been attached. Due to its programmable nature the specification of supportable terminal equipment is rather flexible, and includes provisions for all available AT&T Teletype equipment including Models 28/32 (45.45) baud), 33/35 (110 baud), and 37/38 (150 baud) operating at speeds to 150 words-per-minute. Also supportable is all announced IBM remote terminal equipment capable of operation on the switched-telephone network at speeds to 2000 baud, as well as certain special-purpose data transmission equipment including that intended to interconnect remote computer terminals and the MTS system.

All of the Data Concentrator equipment is sponsored by the Advanced Research Projects Agency (ARPA) in support of the CONCOMP (Research in Conversational use of Computers) project. Usage by the general user population is possible after inquiry to the CONCOMP project and approval by the Project Director, F. H. Westervelt, on an as-available basis. Initially, the following terminals will be supported (about 1 December 1967): AT&T Model 33/35 Teletype, IBM 2741 Typewriter Terminal, and remote computer terminals utilizing AT&T 201A serial synchronous data sets. Operating procedures for use with these terminals are discussed below. At some future date, about 1 January 1968, AT&T X403A Touch Tone digit receivers and 801C3 automatic calling units will become available to MTS users. Other data set equipment may be added by arrangement with the Computing Center and the ARPA project.

MTS Interface

The Data Concentrator appears to MTS as a collection of 32 devices, each of which operates independently of all the others. One or more of these MTS devices may be logically connected to one or more data set circuits for use by the remote terminal. The MTS devices serviced by the Data Concentrator are all of type PDP8 and are identified by names of the generic class "DCXX", where XX is a two-digit decimal number in the range 00 through 31. These names can by used anywhere in MTS that a device name is meaningful. Within the Data Concentrator itself all of these devices, and, in addition, all of the attached data set circuits, appear as a homogeneous collection of devices. Each of these Data Concentrator devices is assigned an internal logical device number in the range from 00 through 77 (octal) as indicated in Figure 1. Each device may be used both as a

source and a sink without regard as to whether it is capable of full- or half-duplex operation.

Generally, message formatting and error control are transparent to the use of a device by MTS, and each such device can be characterized by the ability to transmit an indefinite number of records (of 255 characters or less) in one direction without regard to traffic in the other direction. Although some of the data sets and terminal equipment can operate in a full-duplex mode, that is in both the MTS-outbound and MTS-inbound directions simultaneously, each of the 32 devices interfaced with MTS and, indeed, MTS itself can operate in only a half-duplex mode, that is in only one direction at a time. If a particular device is constrained to half-duplex operation, then the inbound and outbound traffic is interleaved on a record-by-record basis by buffering the messages and turning the transmission circuit end-for-end as required.

A pair of Data Concentrator devices can be interconnected by a copy operation, which assigns to each source device a sink device to which all message traffic produced by that source will be directed. Although only one sink may be assigned to each source by such an operation, any number of sources may produce traffic to the same sink. In this case the traffic is interleaved on a record-by-record basis. Those particular copy operations which define the configuration of the Data Concentrator at any moment are changeable by commands directed to the Data Concentrator itself. In addition, certain commands can cause the behavior presented by any one data set circuit to be altered to fit a particular terminal. In this manner one set of data set equipment can serve terminals of different types, all of which require the same data set type.

Although it is possible to associate a copy operation between any two devices and to change the configuration of any data set from any terminal, and for that matter from MTS itself, the obvious dangers associated with the use of this feature require enforcement of a policy that all such operations will be supervised by special MTS library programs. For the present, the equipment will be operated in a fixed configuration providing service to those IBM, AT&T, and special terminals as described in a later section.

FIGURE 1:

LDN (OCTAL)	DATASET TYPE	TERMINAL TYPE	FUNCTION	PHONE NUMBERS
0	---------	---------	DUMMY FILE	------
1	103	ASYNCHRONOUS; FULL DUPLEX KEYBOARD (SOURCE) PRINTER (SINK); MODEL 33 TTY (110 BAUD)	MASTER CONSOLE (4-9526)	4-4496
2	---------	HIGH-SPEED PAPER TAPE READER (SOURCE) PUNCH (SINK)	SYSTEM RESIDENCE	------
3	---------	---------	UTILITY FILE	------
4 THRU 7	201A	SERIAL-SYNCHRONOUS; HALF-DUPLEX; (2000 BAUD)	HIGH-SPEED DATA TRANSMISSION BETWEEN A REMOTE COMPUTER OR GRAPHICS TERMINAL AND MTS	4-4481 4-4482 4-4399 4-6334

FIGURE 1: (Continued)

10 THRU 17	103A	ASYNCHRONOUS; FULL DUPLEX; KEYBOARD (SOURCE) PRINTER (SINK); MODEL 33/35 TTY OR IBM MODEL 2741 (UP TO 200 BAUD)	LOW-SPEED DATA TRANSMISSION BETWEEN A REMOTE TYPE-WRITER TER-MINAL AND MTS	4-0200 THRU 4-0207
20 THRU 27	103A/202C (103'S ARE DESCRIBED ABOVE, INFO HERE IS FOR 202'S)	ASYNCHRONOUS; HALF DUPLEX (1200 BAUD)	MEDIUM SPEED DATA TRANS-MISSION BETWEEN A REMOTE COM-PUTER, GRAPHIC TER-MINAL, OR PAPER TAPE/ CARD DEVICE AND MTS.	4-4208 4-4297
30 AND 31	X403A	ASYNCHRONOUS; HALF DUPLEX; 10 CHARACTERS PER SECOND [TWELVE BUTTON TOUCH-TONE TELEPHONE RECEIVER; AUDIO RESPONSE UNIT]	LOW-SPEED REMOTE TER-MINAL FOR TRANSMISSION TO AND AUDIO RESPONSE FROM A CEN-TRAL COM-PUTER.	4-4494 4-4483

FIGURE 1: (Continued)

32 THRU 36	801C3 ACU (AUTOMATIC CALLING UNIT)	ASSOCIATED WITH A TOUCH-TONE TELEPHONE RECEIVER	USED FOR MAKING A PHONE CALL UNDER COMPUTER CONTROL; TO INITIATE COMMUNICATIONS BETWEEN A 403 DATASET (TOUCH-TONE RECEIVER) AND THE CENTRAL COMPUTER	(NOT ASSIGNED)
37	---------	---------	--------	(NOT ASSIGNED)
40 THRU 77	IBM 360/67	MTS DEVICE NAMES DC00 THROUGH DC31	DIRECT LINES OF COMMUNICATION BETWEEN THE MULTIPLEXOR CHANNEL AND THE DATA CONCENTRATOR	------

Message Formatting

All transmission between devices within the Data Concentrator is according to the ASCII code interpretation, with translation to that code peculiar to any one terminal type provided in the Data Concentrator and MTS itself. Thus, an automatic translation is effected between the MTS EBCDIC code and the Data Concentrator ASCII code as appropriate and unless explicitly modified (see below). In addition the prefix characters transmitted by MTS will be forwarded to the device unless explicitly modified. Since the internal file structure of MTS permits only those records up to 255 characters in length, any longer inbound message will be automatically split into several records as necessary. In most cases the splitting operation is transparent in the converse outbound operation.

A message between any pair of devices attached to the Data Concentrator is constructed of a string of characters chosen from all those 8-bit characters with the exception of certain control characters defined as in Figure 2. Certain devices cannot recognize some of these characters, and in such a case those characters may be part of the transmitted data stream. This particular choice of control characters reflects current ASCII-8 standards with the provision that all control characters are constrained so that the high-order three bits are (100). This convention is adopted for convenience in the transmission of certain tape formats popular with remote computer terminal equipment; and, in particular, allow transmission of a message consisting of arbitrary 7-bit bytes if the high-order eighth bit is forced to a zero.

The Data Concentrator resident supervisory program recognizes those characteristics peculiar to each of its attached devices which are logically equivalent to an occurrence of one of the control characters. By convention all of these characters, except the SOH and STX, end the record of which they are a part. The ETB, ETX, and EOM are equivalent within the Data Concentrator but have various interpretations to the particular devices involved. The ENQ and EOT may occur asynchronously with traffic between any pair of devices and are interpreted as the attention and the halt conditions respectively. Both the ENQ and the EOT have the affect of purging all buffers involved and forcing device resets. The ACK and NAK are used in connection with circuit supervision and error recovery. The SOH is used to precede a header, which is processed by the transmission equipment itself (see below). The STX is used to precede a message under certain conditions.

Name	Function	ASCII	EBCDIC	TTY	2741
SOH	start of header	001	01	ctl-A	!A
STX	start of text	002	02	ctl-B	!B
ETX	end of text	003	03	ctl-C	RETURN
ETB	end of text block	027	17	ctl-W	!W
EOM	end of file	033	1B	ctl-Y	¢
ENQ	enquiry	005	05	ctl-E (WRU)	ATTN
EOT	end of transmission	004	04	ctl-D (EOT)	!D
ACK	positive acknowledge	006	06	ctl-F (RU)	!F
NAK	negative acknowledge	025	15	ctl-U	!U
DLE	literal-next	020	10	ctl-P	!
	line delete			RUBOUT	#
	character delete			<—	backspace

Figure 2

USE OF DATA CONCENTRATOR BY MTS JOBS

Since the operation of the Data Concentrator is constrained in a fixed-configuration mode from the viewpoint of the user, the internal copy associations and device characteristics will not normally be modifiable except on special request. The initial configuration will include support for some quantity of each of the following terminals as master source/sink to an MTS job: IBM 2741 Typewriter Terminal, AT&T Model 33/35 Teletype, and serial-synchronous transmission circuits to remote computer terminals. Support for some quantity of each of the above terminals will be available for use as a utility device within MTS. These are obtained by any MTS job as the result of the specification of one of the device names assigned to the Data Concentrator (see above) in an I/O operation.

MTS-170-0

12-1-67

A MTS-inbound or MTS-outbound record beginning with the SOH character is directed to a small command language interpreter which is embedded in the device support routines associated with the MTS devices. Such a record is constructed within MTS by prefixing the SOH character to one of the command lines described below. The same record can be constructed at the terminal by prefixing the STX-SOH code combination to one of the command lines described below. The first STX is necessary in the latter case to differentiate the MTS - interpreted commands from those interpreted within the Data Concentrator itself. Commands recognized at present include:

JOB - Print MTS job number and device name for this I/O device.

PFX OFF - Turn off prefix character transmission from MTS. (equivalent to @SP modifier).

PFX ON - Turn on prefix character transmission from MTS (equivalent to @¬SP modifier).

BIN ON - Turn off both EBCDIC/ASCII code translation and prefix character transmission from MTS (equivalent to @BIN modifier).

BIN OFF - Turn on EBCDIC/ASCII code translation (equivalent to @¬BIN modifier).

These commands are effective only if the MTS job is in condition to read from or write on the device in question. If a command is not recognized following the SOH the entire command line is deleted.

A reference to an MTS device name of type PDP8 may be followed by modifiers of the form @MOD or @¬MOD (see elsewhere, this manual). Those modifiers recognized in an unusual manner in connection with the Data Concentrator are:

@SP - Turn off prefix character transmission from MTS (equivalent to PFX OFF).

@¬SP - Turn on prefix character transmission from MTS (equivalent to PFX ON).

@BIN - Turn off both EBCDIC/ASCII code translation and prefix character translation from MTS (equivalent to BIN ON).

@¬BIN - Turn on EBCDIC/ASCII code translation (equivalent to BIN OFF).

Other modifiers applying to the MTS device itself are given the standard meaning.

AT&T Models 33/35

The operation of the Teletype itself is outlined elsewhere; only those features differentiating the Data Concentrator operation from the conven-

tional operation are summarized here. Operation of the Teletype is in a full-duplex mode, so that separate data streams inbound to MTS from the keyboard and outbound to the printer from MTS are possible. As characters are passed from the keyboard to MTS they are echoed (repeated) on the printer for operator verification unless the printer is active for some MTS outbound function, in which case the accumulated echo characters are saved until the end of the active line and then output on the printer.

The end-of-line code is the return key. MTS will respond by sending a carriage-return/line-feed to the printer. The end-of-file code is the control-Y key. MTS will send a backslash and then respond as in the end-of-line case. An attention is transmitted using the BREAK key (or the WRU key on Teletype equipment not provided with break feature) MTS will send a exclamation point and then respond as in the end-of-line case. A character may be deleted by the back-arrow key. A line may be deleted by the RUBOUT key. MTS will respond in the latter case by sending a pound sign "#" followed by a carriage-return/line-feed to the printer. See Figure 2 for additional codes.

With some care it is possible to achieve a rather efficient utilization of the Teletype terminal connected to the Data Concentrator. First, due to a one-line print buffer maintained in the Data Concentrator, the MTS output function is rather more efficient than that used with the standard Teletype terminals. However, due to this feature, the response time for an attention interrupt to be acknowledged on the printer will be delayed by the time it takes to empty this buffer. Second, due to a one-line input buffer maintained in the Data Concentrator, it is possible to queue up to 72 characters for presentation to MTS before the MTS program calls for them. Thus it is possible to enter a command to MTS while output is being produced on the printer or while MTS is processing a previous command. If this buffer overflows, echoing of the input characters will cease, following which it is necessary to either delete the input line or to backspace one character and enter the end-of-line code. The attention operation always purges this buffer concurrently with presentation of the attention condition to MTS.

IBM 2741 Typewriter Terminal

Operation of the IBM 2741 is similar to that in connection with the 2702 transmission control (see elsewhere, this manual). Although the 2741 is basically a half-duplex device, it is possible to enter a new command to MTS while a previous command is being processed. When the printer is active, use of the keyboard can be requested by the attention key. At the conclusion of the current line of output the keyboard is unlocked. Use of the attention key when the keyboard is unlocked causes an attention interrupt to MTS.

Since this terminal operates in a half-duplex mode, no echoing operation is possible. A character may be deleted by the backspace key, and a line may be deleted by the pound sign "#". The end-of-line code is the return key and the end-of-file code is the cent sign ("¢") key. MTS will respond

to either the line-delete, end-of-line, or end-of-file keys with a carriage-return/line feed to the printer. See Figure 2 for additional codes.

Remote Computer Terminal Transmission Facilities

Interconnection between MTS and a remote computer terminal is possible using any data set attached to the Data Concentrator. Using the 103A data set normally associated with the AT&T and IBM keyboard/printer terminals, rates to 200 baud can be sustained in a full-duplex environment with start/stop codes to 12 bits in length (including start and stop bits). Using the 202C data set, rates to 1200 baud can be sustained in a half-duplex environment with codes similar to those of the 103A. The fastest transmission rate obtainable on the switched telephone network at this time is provided by the 201A data set. Using this data set a rate of 2000 baud is sustained in a half-duplex bit-synchronous environment with codes to 12 bits in length.

Only the 201A is supported initially by the Data Concentrator resident supervisor. This equipment is operated with an eight-bit byte size including no vertical parity bit. Longitudinal parity control is provided with a 16-bit cyclic checksum in the form of two bytes transmitted immediately following the end-of-record (ETX, ETB) characters. All transmitted records involving header or text transmission are acknowledged by the intended receiver in the form of a single-character message, which may be embedded within header or text information proceeding from the intended receiver back to the transmitter.

A message is transmitted, after a short 30 millisecond synchronization delay, at a rate of 250 characters per second. Data set turnaround for acknowledgement purposes is accomplished in 20 milliseconds within the on-campus Centrex telephone network and in 200 milliseconds outside the immediate vicinity of Ann Arbor. This longer delay is necessitated by certain repeater equipment used on intercity trunks.

Program support for the matching remote computer terminal can be provided in either of two ways. If the remote machine is a PDP-8 or a PDP-7 then an opportunity exists to use a scaled-down version of the resident multiprogramming supervisor used in the Data Concentrator itself. At least three such versions are now in current University use in systems which include cathode-ray display equipment, card readers, high-speed paper and magnetic tape equipment, RAND tablets, direct-access devices, and audio switch equipment.

The scaled-down Data Concentrator supervisor is designed to operate with 201A interface equipment specially constructed for the purpose. Several interfaces of this kind have been constructed for use at various campus locations. A rather efficient remote batch-entry and terminal concentrator can be constructed from these components attached to a PDP-8 running with the multiprogramming supervisor. Both the supervisor and the 201A interface are described in references obtainable from CONCOMP project offices.

Alternatively, support for the remote machine can be developed for special purposes by using the message formatting and transmission protocol described in the references cited. These conventions have been established to provide secure error recovery procedures consistent with reasonably-sized support programs. Interface hardware can consist either of the interface mentioned above or can be specially synthesized for the purpose. With care, it is possible to connect an IBM SDA-II "blind pig" Transmission Adaptor or any other IBM Binary Synchronous Communications equipment to the Data Concentrator with suitable program support.

UMMPS AND MTS: A GENERAL DESCRIPTION OF THE OPERATING SYSTEM

UMMPS

UMMPS (University of Michigan Multi-Programming System) is a multiprogramming operating system for the IBM System/360 series computers. UMMPS executes jobs, which are initiated and controlled from the operator's console typewriter. Each job runs in problem state and uses supervisor calls for all its input and output operations.

A job program is the basic set of instructions which are executed when an UMMPS job is run. Job programs are core-resident, along with the UMMPS supervisor and subroutines. A reentrant job program can be executed at the same time by more than one job. When a job program is written a set of device types and a set of memory buffers of various sizes are specified. Corresponding actual devices and memory space are allocated for any job initiated with that job program, and these are retained until the termination of the job. By means of supervisor calls, jobs may obtain and release additional devices and storage space during their execution. A single device (e.g. a card reader, communications terminal, or a disk module) is available for at most one job at any given instant.

The most recent (November, 1967) versions of UMMPS use the dynamic relocation hardware peculiar to the System/360 Model 67 in order to provide a virtual memory buffer space of 256 pages (one page = 4096 bytes) for each job. The supervisor manages real core memory with a demand paging algorithm, using an IBM 2301 Drum for secondary storage. The policies for using the queues are as follows:

(a) A channel program is constructed to read pages from as many of the drum sectors as possible. The remaining sectors, if any, are then used for servicing the first write requests. More than one of these channel programs may be constructed at one time and chained together, providing a complete schedule of drum operations for the next several revolutions of the drum.

(b) Unless a page in real core enjoys a special "temporarily resident" status, it is placed on the queue for page-out as soon as it enters core. This queue will be reordered according to the usage of the pages enqueued, by means of an algorithm expected to undergo a great deal of modification and adjustment.

(c) When the paging drum processor requests a set of pages from the supervisor to be written on the drum, those pages are made available from the head of the page-out queue only in case core is sufficiently full.

(d) The supervisor may also refuse to supply the drum processor with an empty page for a page-in request, if there are almost no available empty core pages.

The main CPU queue is a list of jobs which are scanned in top to bottom order, and are given the CPU if they are ready to use it. Whenever a job initiates an I/O operation which makes it not ready for execution and guarantees that an interrupt will occur when it again becomes ready, its entry is removed from the main CPU queue. Other not ready jobs, e.g. those in terminal wait or in wait for a non-interrupting event, remain in the CPU queue and are passed over when their turns come. (A job in execution can also voluntarily place itself at the bottom of the CPU queue.)

When an interrupt occurs for a job which does not have an entry on the main CPU queue, this job immediately preempts the job currently allocated the CPU (i.e. an entry is placed on the very top of the queue and the new entry is dispatched, with the current job remaining interrupted). Each job runs until it waits for some event or until a timer interrupt signals the end of a time slice.

Each job has a personal CPU queue which is used to keep track of multiple levels of execution. The top entry refers to the sub-task (if any) currently in contention for the CPU. A lower-level entry represents a sub-task which has been interrupted but may later be resumed. For example, some I/O interrupts cause a new entry on the queue for their processing. And a job may be given an attention interrupt by the operator (or MTS user), and then later restarted.

A wait queue is maintained for each job which contains an entry for each entry in the personal CPU queue which represents a sub-task currently waiting. Thus a job may be waiting at several of its lower levels, executing at the top level, yet an interrupt signalling the end of such a wait can be properly recorded by removing a wait queue entry for the appropriate CPU queue level.

MTS

MTS (Michigan Terminal System) is a reentrant job program in UMMPS. It provides the capability of loading, executing and controlling programs from remote terminals and through a batch stream. Together with UMMPS, MTS provides a simple but powerful time-shared computer system, whose salient features are these:

(a) Command language. Several dozen commands are available to cause the running and monitoring of programs, the manipulation of line files, and other communication with the system.

(b) Line files. A system of information organized in units of lines (1 to 256 characters) and files (0 to many thousands of lines) is provided for the storage of programs and data. A file

may be public or private, and a private file may be permanent or temporary. These files reside on direct-access storage devices.

(c) Logical devices. When an MTS user specifies the origin or disposition of data, he may give, interchangeably, the name of a file location or a physical device. A logical device name or number is then attached to it. It may refer, for example, to a system (public) file, a new temporary private file, a card punch, or the operator's console.

(d) Dynamic loader. A program to dynamically load programs is an integral part of MTS. It may be invoked by both commands and subroutine calls.

(e) Libraries. External symbols which have been referred to by a set of loaded programs, but not defined, may be resolved by reference to a private or system library, which is a file containing object programs in a special format. Facilities exist in MTS and the Loader to pass over a library and selectively load only needed subroutines (and the subroutines that they need, etc.).

(f) Language processors. The MTS system makes available the IBM System/360 F-Level Assembler, the IBM FORTRAN IV G-Level Compiler, WATFOR (University of Waterloo Fortran load and go compiler), PIL (Pitt Interpretive Language), SNOBOL4 (a string manipulation language), and UMIST, a string processor based on the TRAC text-processing language. These programs reside in system files, and are executed in the same way as one's own programs produced by their execution. Other powerful system features, such as the IOH/360 input-output conversion subroutines, macro libraries, plotting routines, etc., reside in the library and other system files.

EXTERNAL SPECIFICATIONS

This major section of the manual presents external specifications: those details the user needs to know to use the system.

FILES AND DEVICES

FILES

A FILE is an ordered set of zero or more lines where each line consists of from 1 to 255 characters. Each line has associated with it a line number which is not part of the line. Externally a line number is of the form "snnnnn.nnn" where "s" is a sign ("+" or "-") and "n" is a decimal digit. Leading "+" signs and zeros, trailing decimal points, and trailing zeros after the decimal point may be omitted. Internally in calls to input/output subroutines (See MTS-252) a line number is a full-word binary integer the value of which is 1000 times the external form of the line number. For example the line number whose external form is 1.000 (which is the same as 1) is stored internally as

000003E8[hex] (=1000[decimal])

A line number may also be written externally as LAST or LAST ± line number, where LAST stands for the line number of the last line currently in the appropriate file (if the file is empty, the value is zero).

There are two methods of reading and writing a file: indexed and sequential. An indexed I/O operation is one in which the line number is specified explicitly in the call. A sequential I/O operation is one in which the line number is determined by the system as follows: If the first reference to a file is sequential the line number is the initial line number specified when the file name was entered (or 1 if none was given). For a sequential read that is not the first operation the line number is taken from the line immediately following the last line read or written in a file. For a sequential write that is not the first operation the line number is found by adding the increment specified when the file name was entered (or 1 if none was given) to the line number of the last line read or written.

When a file is created (see MTS-230/CRE) a size may be specified for it and an amount of space sufficient to hold a file of this size is allocated on some direct access device. If an attempt is made to put more lines into a file than will fit in the space allocated to it, the system will try to allocate more space for the file. If it is not possible to allocate more space, the system will indicate that the size of the file has been exceeded and will refuse to put more lines in it.

IMPLICIT CONCATENATION

Although the maximum size of any single file is limited to approximately 7000 to 12000 lines, the amount of data that may be referred to by a given

file name is effectively unlimited because several files may be automatically chained together in two ways. Whenever a line beginning with the character string "$CONTINUE WITH FDname", is read from any file or device, reading continues with the file or device FDname and the $CONTINUE WITH line is not passed to the caller. (See MTS-230/SET for a way to override the action.) This is called implicit concatenation. The other method of chaining several files together is explicit concatenation which is described below in this section.

FILE NAMES

File Names are the names of logical files kept on the disk. There are three types: user files, scratch files, and library files. The name of a user file consists of from one to twelve characters. If more than twelve characters are given, the first twelve are used. The name cannot contain blanks, commas, or left parentheses, plus signs, or semicolons, and any lower-case letters are automatically translated to their upper-case equivalent. It is strongly recommended that names assigned be alphanumeric only, i.e., no special characters, since the specification may change in the future. Scratch file names consist of from two to nine characters the first of which is a minus sign. They are automatically created the first time they are mentioned and are destroyed when a user is signed off. Library file names consist of an asterisk followed by from one to fifteen characters the last of which is not an asterisk. In use, user names are prefixed by the four character signon I.D. to make them unique. Library files may be accessed by users but are password protected against modification. Since it is possible to confuse file names and device names (see below), there must be some way to indicate that a name is a file and not a device. This is done by putting a # before the name of the file. The # is not part of the name.

DEVICE NAMES

Device Names are the names of physical devices as recognized by the UMMPS supervisor. The names are 4 characters long (padded with trailing blanks, if necessary). If any of these names is used as a FDname it is assumed to refer to the corresponding device unless the name is prefixed by a #. In general pseudo-device names (see next section) are to be used instead of device names when physical devices must be referred to. Some of the device names currently recognized by UMMPS are:

Device Names	Device Type
DT1	DISP
DC00 through DC31	PDP8

PCH1	PCH
PCH2	PCH
PCH3	PCH
PTPR	PTAP
PTPP	PTPP
PTR1	PTR
PTR2	PTR
PTR3	PTR
RDR1	RDR
RDR2	RDR
RDR3	RDR
T0C0	7TP or TP
T0C1	7TP or TP
T0C2	9TP or TP
T0C3	9TP or TP
T0C4	7TP or TP
T0C5	9TP or TP
T0C6	7TP or TP
T0C7	9TP or TP

In addition to these, MTS recognizes the name "OPER" as the name for the operator's console.

PSEUDO-DEVICE NAMES

Pseudo-device names are synonymns for other file or device names which are assigned for a given user of MTS. They consist of an asterisk followed by from two to fourteen characters the last of which is an asterisk. Seven of these names described below are automatically defined for each user and additional ones may be defined using the program *MOUNT (see MTS-160). The predefined pseudo-device names are:

DUMMY

The name *DUMMY* has a special meaning. It may be used as any FDname. For output files or devices it represents an infinite wastebasket: lines are accepted and they disappear. For input files or devices it represents an empty data set: every time a line is requested an end-of-file condition is returned.

SOURCE

The name *SOURCE* has a special meaning. It may be used as any input FDname. It refers to the current source file or device.

SINK

The name *SINK* has a special meaning. It may be used as any output FDname. It refers to the current sink file or device.

AFD

The name *AFD* has a special meaning. It may be used as any FDname. It refers to the current active file or device (that file or device which was obtained by $GET or $CREATE).

PUNCH

The name *PUNCH* has a special meaning in batch mode. It may be used as any ouput FDname. It refers to the system card punch established for that batch run.

MSINK

The name *MSINK* has a special meaning. It may be used as any output FDname. It refers to the master sink file or device which is the terminal for conversational operation and the printer for batch operation.

MSOURCE

The name *MSOURCE* has a special meaning. It may be used as any input FDname. It refers to the master source, which is the terminal in conversational operation and the operator's console in batch operation. (It should not normally be used in batch operation.)

MODIFIERS

The action of a file or a device may be changed by appending to the file or device name one or more modifiers. Each modifier consists of an at-sign ("@") [mnemonic for "attribute"], followed by the modifier name as given in the following table. (For meanings of the modifier bits and further explanation, see section MTS-252.) A modifier name may be preceded by a

not-sign ("¬") to reverse its meaning. (Teletype users should use a minus sign). If any modifier is given explicitly in a call to an I/O subroutine (see MTS-252) the setting given in the call overrides the setting given with the FDname.

Name	Bit Set
S	31
SEQUENTIAL	31
I	30
INDEXED	30
EBCD	29
BIN	28
BINARY	28
LC	27
LOWERCASE	27
UC	26
CASECONV	26
NOCC	25
NOCARCNTRL	25
CC	24
CARRIAGECONTROL	24
SS	24
STACKERSELECT	24
PFX	22
PREFIX	22
PEEL	20
GETLINE#	20
RETURNLINE#	20
MCC	18
MACHCARCNTRL	18
TRIM	16
SPECIAL	14
SP	14

Examples of use:

```
RDR1@BIN
FILE23@I@UC
```

LINE NUMBER RANGES

When files are written or read indexed, the line number is always explicitly given. However, for sequential read and write of files, the question of the starting and stopping line numbers and the line number increments arises. If no specification is made, for sequential read the beginning line number is 1 and the ending line number, (the one that forces an "end of file" condition if exceeded) is 99999.999. Similarly for sequential write the default beginning line number is 1 and increment is 1. If for a given usage a different set of specifications is wished, a line number range specification may be attached by following the name (or name - modifier combination) with the construct:

```
(b,e,i)
```

where b is the beginning line number, e is the ending line number, and i is the increment. Applying no range specification is equivalent to

```
(1,99999.999,1)
```

B, e, and i, like any line number reference, may each have one of three forms

```
linenumber
LAST
LAST±linenumber
```

For the last two forms, the value of LAST is the line number of the last line in the specified file. If the file is empty, LAST has a value of zero.

Any combination of these three items may be left out: trailing commas resulting from the omission of items may be left off; leading and internal commas are required.

Examples are:

```
FILE(20)
FILE2(20,30)
NAMEEMAN(-1.359,477.2,.01)
OUT@MCC(322)
OUTT@PFX@CC(10,,10)
FYLE(LAST)
PHYLE(LAST-4,LAST-1)
PHILE(LAST-10)
```

Since devices are treated like sequential files--line numbers are faked for them--the above may also be applied to devices.

EXPLICIT CONCATENATION

Several files and/or devices may be chained together by using explicit concatenation. This is done by giving the names of the files and devices with their modifiers and line number ranges in the order desired separated by plus signs. For example

```
-LOAD(1,100)+*SSP
```

means the contents of lines 1 through 100 of -LOAD followed by the contents of *SSP. If two or more consecutive names in an explicit concatenation are the same, all but the first may be omitted. For example

```
A(1,1)+(LAST)
```

is the same as

```
A(1,1)+A(LAST)
```

If one or more members of an explicit concatenation uses implicit concatenation (i.e., contains $CONTINUE WITH FDname) the entire implicit concatenation is used as that portion of the explicit concatenation. In this case the FDname in the $CONTINUE WITH may itself be an explicit concatenation. The processing of the next member of an explicit concatenation is started whenever a return code of 4 (end-of-file) is received on a read operation or the ending line number of the current member is exceeded on a sequential write operation. Care should be exercised when using indexed operations on a concatenated file.

If some member of an explicit concatenation does not exist or is not available and the user is not running in batch, he will be given a chance to enter a replacement name the first time this member is used. This replacement name may be any explicit concatenation of files or devices and replaces only the one member of the original concatenation.

Some examples of explicit concatenation are:

```
MAIN+SUBR
DATA(1,1)+(3,10)@UC
*SOURCE*+DATA+ALLOC(1,10)
*TAPE*(1,10000)+*TAPE2*
```

USAGE

File names and device names are interchangeable. That is, a file name may in general be used anywhere a device name may appear, and the converse. The generic quantity

```
<FDname>
```

used in these writeups will signify such an interchangeable designation which may be an explicit concatenation of file and device names, and

<Fname>

or

<Dname>

will be used if cases arise where a restriction to a single (i.e., no concatenation) file or device name (respectively) is necessary.

The above interchangeability implies that those names which are listed above as device names are preempted from being file names. If it is desired to specify that a name is to be taken only as a file, the name should be prefixed with a #. For example,

RDR1 refers to reader 1
QQSV refers to file QQSV
#RDR1 refers to file RDR1
#QQSV refers to file QQSV

INPUT LINES

In the normal course of processing, lines are input from the current source file or device and acted upon by the MTS monitor. In addition, if the current source is not the user's console and the console has an attention feature, source can be restored to the console.

An input line to the MTS monitor is either a command or a data line. If the first character is a dollar sign ($) and the second character is not a dollar sign, the line is a command line; otherwise it is a data line.

COMMANDS

A command line starts with a dollar sign, immediately followed by the command. No blanks are allowed. The command may be followed by zero or more parameters, depending on the command. These parameters are set off from each other and the command by one or more blanks. Commas, semicolons, equal signs, and parentheses are also used as break characters in various places. A description of each command follows this section.

DATA LINES

Data lines are put into the current active file on disk. If the user has not established an active file, he will be so notified when a data line is sensed. If automatic line numbering is off, the line is inspected for a line number, and what follows the line number is put in the file as the line with that number. If there is no line number, zero will be assumed. If automatic numbering is on, the line is not inspected but is taken verbatim, except that, if the first two characters are dollar signs, only one is transmitted to the file. See section MTS-240 for a description of the format of data lines.

PREFIXING

So that the user can know "who is speaking" and so he knows when input is expected, the first character of all lines on consoles (not card readers, punches, or printers) except the operator's 1052 (device name OPER) is a special prefix character. On output lines this is typed ahead of the message. When input is requested, either the prefix character

(automatic numbering off) or the prefix character followed by the line number (automatic numbering on) is typed at the front of the line. The prefix characters are:

issued by MTS monitor:	#
issued by user's program at run time:	blank
issued during loading:	.
issued during LIST or COPY:	>
issued to prompt user for reply:	?
issued by the PIL interpreter	=
issued by SDS (Symbolic Debugging Sys.)	+

EDITING

To allow ease in using the system, means have been devised to allow the user to delete the preceding character (i.e., backspace and erase), delete the whole line and start over, tab to a given column, tell the system there is no more data (EOF), and so on. In addition, the user needs some way to indicate that "this is the end of the current line". This is done by having a particular character indicate each function, such as ? for line-delete. To allow the user to actually feed in these control characters as data, a literal-next character is defined, which says that the following character is not to be interpreted as a control character. For example, if ? means line-delete, and ! is the literal-next character, then ! ? would be considered as the single text character ? (and of course, to get a ! into the text, !! must be typed).

Due to the different character sets available on different terminals, which character represents which function varies from device to device. Also, due to differences in transmission controllers attached to the 360 (2701, 2702, and Data Concentrator), a single terminal will behave differently depending which phone number is called. A table of control characters for devices currently supported as terminals is listed below. (This is a summary of the information presented in the user's guides in Section 100 of this manual. See those writeups for details.)

Terminal	End line Character	Delete Line Character	Delete Char. Character	Literal next Character	Cause Attn.	Cause EOF
1052-OPER	RETURN EOB	?	"	(none)	GOOSE JOB	CANCEL
TTY (THROUGH	X-OFF 2702)	CONTROL-N	CONTROL-A	CONTROL-Z	BREAK	CONTROL-C
TTY (THROUGH DATA CONCENTRATOR)	RETURN	RUBOUT	<-	CONTROL-P	BREAK CONTROL-E	CONTROL-Y
2741 (THROUGH	RETURN 2702)	ATTENTION	BACK-SPACE	!	ATTN	¢
2741 (THROUGH DATA CONCENTRATOR)	RETURN	#	BACK-SPACE	!	ATTN	¢
1050 (2702)	RETURN or EOB	EOT	BACK-SPACE	!	RESET LINE	¢

CONTINUATION

If the last character in the source stream (prefix char #) line is a minus sign ("-"), then the next input line is assumed to be a continuation. Continuation begins with the first character of the next line, which may be assumed to replace the "-" continuation character in the previous line. As many continuation lines as desired may be used, with the restriction that their total length may not exceed 255 characters. This is effective only for lines read by the MTS monitor, i.e., read when the prefix character is #.

LIMITS SPECIFICATION

In order to prevent run-away jobs in batch runs, it is necessary to provide some means for the user to limit the time a job may use and the amount of output (paper and cards) it produces. It is also often desireable for users at a terminal to be able to limit a RUN of a program. To provide this facility, limit keyword-parameters are provided. They are of two types, global and local.

Global limits are limits for the entire job, from SIGNON to SIGNOFF. They are to be placed on the $SIGNON card for batch jobs. Default values are assumed for any omitted parameters. The current default values are given in the table below.

Within the global limits imposed at SIGNON, the user may specify separate local limits on any RUN, LOAD, RESTART, or START command. If a RESTART or START command specifies no limits, then what is left of the limits specified by the orginal RUN or LOAD command is used.

The first limit to be exceeded, whether global or local, causes a comment to be printed and the job to be terminated (in batch operation), or a return to command mode (in terminal operation).

QUANTITY	KEYWORD PARAMETER PROTOTYPE	UNITS	GLOBAL DEFAULT	EXAMPLES
CPU Time	{T } { } {TIME}=n{S} {M}	seconds seconds minutes	10	T=6 TIME=3.27S TIME=.1M
Pages of Output	{P } {PP }=i {PAGES}	pages	50	P=10 PAGES=62
Punched Card Output	{C } {CP }=i {CARDS}	cards	1	C=12 CARDS=400

where i is an integer of up to 5 digits.
n is a number similar in form to a line number: up to 5 digits in front of the decimal point; up to 3 digits after the decimal point.

COMMANDS

The following subdivisions of section 230 describe the commands available. For each is given the name, a prototype explicitly giving the syntax, descriptions of the purpose, usage, and effect, and examples.

For most commands, the first three letters suffice to identify it, so only the first three need to be given. For those commands which may be truncated, the unnecessary part will be identified by underlining in the prototype.

Notation conventions used in the prototype are: lower case represents a generic type which is to be replaced by an item (such as a file name) supplied by the user. Upper case is to be repeated verbatim in the command (although commands, and those parts repeated verbatim, can be upper-case, lower-case, or mixed.) Material in brackets [ ] is optional. Material listed in braces {} represents choices, from which exactly one must be selected. Three dots following a syntactic unit indicates it may be repeated indefinitely.

COMMAND DESCRIPTION

Name: ALTER

Purpose: To alter the contents of a general register, floating point register, or specified core location(s).

Prototype: $ALTER pari parj ...

where pari specifies what is to altered and may be any of the following:

GRx	where x is a decimal (or hexadecimal) integer from 0 to 15
FRy	where y is one of the integers 0, 2, 4, 6
[{RF=HHHH}] HHH {RF=GRx }	where RF=HHHH specifies an optional core storage relocation factor and HHHH specifies the displacement from the relocation factor. The relocation factor may be specified as RF=GRx in which case the contents of GRx is used.

parj specifies the new contents of the register or location(s) to be altered and may be any of the following:

C'XXXXX'	character constant expression
X'HHHHH'	hexadecimal constant expression
HHHH	free form hexadecimal constant
F'YYYYY'	fullword constant expression
H'YYYYY'	halfword constant expression

Usage: Any number of items may be altered each time the command is given. This requires the specification of one element from the pari list and one from the parj list for each item to be altered. The default value for relocation factor is zero (unless otherwise set by the $SET command); however, once the user specifies a relocation factor, that value remains in effect until a subsequent "RF=HHHH" appears in the command line.

Fffect: Register numbers and addresses are checked for validity and complaint is made it illegal values are specified.

Content parameters are treated as follows:

1. Character constants are placed, left justified, in the register or location specified. Truncation or padding with trailing blanks is applied whenever appropriate for registers. Sixty-four bit length is implied for floating point registers. Any EBCDIC character including blank may be given between the delimiting primes; a prime in the character string must be represented by two consecutive primes.

2. Hexadecimal constants (whether free form or in the format X'HHHH') are treated differently for general registers, floating point registers and core locations. For general registers there may be from one to eight hexadecimal digits including leading zeroes. The integer value of the constant is loaded into the register specified.

   For floating point registers the constant is placed, left justified with leading zeroes retained, in the register. Truncation (or padding with trailing zeroes) is applied whenever it is appropriate. Sixty-four bit length is implied.

   For core storage locations the hexadecimal digits are placed, two per byte with leading zeroes retained, into consecutive core locations beginning with the byte specified by the current value of the relocation factor and the displacement given in pari. If an odd number of hexadecimal digits is given, the last byte of core storage altered will have bits 4...7 set to zero.

3. Decimal constants whether fullword or halfword consist of a sign followed by the decimal digits all enclosed by primes and preceded by F or H, respectively. The "+" sign is optional; the "-" sign is required. For general registers the number is loaded into the register specified. Decimal constants may <u>not</u> be specified for floating point registers. For core locations the value is loaded into the fullword (or halfword) whose high order byte is specified by the current value of relocation factor and displacement. This is done <u>without regard to boundary alignment</u>.

Examples:

MTS-230/ALT-0

12-1-67

```
$ALTER      GR3     1A3E0     FR6     X'4110A'
$ALT        RF=18AE2     2BE     X'D502CC7E6000' 3E0  X'05EF'   GRA 0
$ALTER      18DA0   C'DON''T DO IT'
$ALTER      RF=18808 0150  1808  02EA  1B97  RF=19600  2B6
$ALTER      RF=1A800  AEC   F'-1000'
```

COMMAND DESCRIPTION

Name: COMMENT

Purpose: To allow insertion of comments.

Prototype: $COMMENT any text

Usage: This command is completely ignored and does nothing at all.

Examples: $COM This is a comment.

COMMAND DESCRIPTION

Name: COPY

Purpose: To copy an existing file into another one.

Prototype: $COPY [FROM] [fromFDname] [TO] [toFDname]

where-

fromFDname is the name of the file or device which is to be copied.

toFDname is the name of the file or device which is to be copied into.

Usage: This command causes the lines of a file or device to be read and inserted into a file or device. I.e., it makes a copy of the file. If an exact copy is wanted (i.e., each line under the same line number as in the original file), then the indexed modifier ('@I') must be appended to toFDname. Note that if indexed is not specified, a renumbering always occurs. Normal usage is to copy a file, but by making toFDname a punch the file may be punched, and by making fromFDname a reader and toFDname a printer or punch an 80-80 list or reproduce is achieved.

If fromFDname is omitted, the active file is assumed and if toFDname is omitted, the current sink is assumed. If fromFDname is omitted or if toFDname precedes fromFDname then the words "TO" or "FROM" must be included. Complaint will be made if either fromFDname or toFDname specify a file that does not exist, or a device that is either not available or is the wrong type (is output device, or input device, respectively). There exists one ambiguous case:

$COPY FROM TO toFDname

In this case, the "TO" is taken as the "noise" word and "FROM" as the fromFDname.

Effect: Starting with the first line in the file, lines are read sequentially from fromFDname until an EOF condition is read. Each line is inserted as is into toFDname.

Examples:

```
$COPY A TO B

$COPY FROM SNARK TO BANDERSNATCH

$COPY FILE1 FILE2
```

```
$COPY F1(1,20)  F2@I
$COPY F1  TO F2(10,,10)
$COPY TO F2
$COPY  TO F2 FROM F1
$COPY F1+F2(1,10)+(25,100) TO F3
```

COMMAND DESCRIPTION

Name: CREATE

Purpose: to create a file, either a private file or a scratch file.

Prototype: $CREATE Fname [SIZE=filesize] [VOLUME=volname]

where-

Fname is the name of the file to be created.

Filesize is the estimate of the size of the file, expressed in the number of 40 byte lines. If no size parameter is given, a small file (approximately 50-100 such lines) is assumed. In estimating the the size of a file, the total number of bytes is important; hence 100 80 byte lines is approximately equivalent to 200 40 byte lines, and so on. The default size of a scratch file is equivalent to SIZE=1300, so that the default size is generally large enough and thus an explicit CREATE is not needed.

Volname is the name of the (presumably private) volume on which the file is to be created. If it is omitted the file will be placed on a public volume where there is space available for it.

Usage: This command is used to create all files including scratch files, except that it is not needed for scratch files unless a SIZE parameter is necessary. It also establishes the file created as the current file. Complaint will be made if a file by that name already exists.

Effect: Fname is created. Fname becomes the current file.

Examples:

```
$CREATE QQSV

$CREATE FIT3STANZA7 SIZE=10
```

COMMAND DESCRIPTION

Name:	DESTROY
Purpose:	To destroy a user's file.
Prototype:	$DESTROY Fname
where-	Fname is the name of the file to be destroyed
Usage:	This is used to destroy user files only, not library files.*files.*files.*Complaint is made if the parameter is missing, the file specified does not exist or is a library file. If the parameter is correct the name of the file and an announcement that it is to be destroyed are given and confirmation is requested. There are three positive confirmations: !, OK, or O.K. Anything else is a negative confirmation. Appropriate response is made to the user after the confirmation indicating that it has or has not destroyed the file.
Example:	$DESTROY AARDVARK

---

*Library files may be destroyed only if the online operator's console issues the command.

COMMAND DESCRIPTION

Name: DISPLAY

Purpose: To display any of the following

1. general registers (any or all)
2. floating point registers (any or all)
3. a specified region of core storage

Prototype:

$DISPLAY [ON aFDname] [par i] par j

where aFDname is the name of the file or device on which the items specified are to be displayed.

par i are parmeters that govern the format of the display of items retrieved from core storage. These may be used to turn format switches on or off according to the following table:

Format Feature	Turn On	Turn Off
hexadecimal conversion	HEX	NOHEX
mnemonic conversion	MNEM	NOMNEM
EBCDIC conversion	EBCD	NOEBCD
EBCDIC conversion	BCD	NOBCD
double spacing	DSPC	SSPC
double spacing	DBLS	SGLS
double spacing	SP2	SP1
long output records (130 characters)	ORL=L	ORL=S
short output records (70 characters)	ORL=S	ORL=L

par j are parameters that govern the content of the display. They may be chosen from the following list:

GRx where x is the character "S" (if all general registers are to be displayed) or a decimal (or hexadecimal) integer from zero to fifteen.

FRy where y is the character "S" (if all floating point registers are to be displayed) or one of the integers 0, 2, 4, or 6.

PSW the PSW at the time of the last error return, interrupt, etc. is displayed.

RF=HHHH where HHHH is a hexadecimal relocation factor for core storage locations specified in single location or block form.

RF=GRx where x is an integer between 0 and 15. The relocation factor is set to the contents of GRx.

HHHH the hexadecimal address of a single core location to be displayed (address is relative to the current relocation factor)

HHHH...HHHH the range of hexadecimal addresses of a core region to be displayed (addresses are relative to current relocation factor). The second address given must not be less than the first.

Usage: Default cases for omitted parameters:

aFDname	sink
par i	ORL=L for line printers, ORL=S for console typewriters
	HEX
	SSPC
par j	The default value for the relocation factor is zero (unless set otherwise by the $SET command).

The only restriction on the order of parameters in the command line is that "ON aFDname" must appear first if it appears at all. Format parameters and relocation factors assume and retain tne default values until a command line entry specifies otherwise. Format parameters and relocation factors may be changed by subsequent entries in the command line. Whenever a content parameter is encountered in the command line it is processed immediately using the format parameters and relocation factor value in effect at that time. If the last item in a command line is not a content parameter, an appropriate comment is made. Hexadecimal entries in the command line may contain a maximum of eight hexadecimal digits (including leading zeroes).

Effect: The parameters are inspected to see that the file, if specified, exists or that the device, if specified, is an output device which is free. If so the remainder of the parameters are inspected in the sequence given and appropriate action is taken. If an individual general register or floating point register, or the PSW is specified, it is displayed in labelled hexadecimal format; all 64 bits of a

floating point register are displayed. Complaint is made if the register number is illegal. Similarly, if all of the general registers (or floating point registers) are requested they are displayed in labelled, hexadecimal format.

Blocks of storage are displayed by calling the subroutine SDUMP. See the writeup on SDUMP for details concerning the formats.

Examples:

```
$DISPLAY  GR3  FRS  EBCD  18E08...18FA6  (see note below)

$DISPLAY  ON  PTR2  MNEM  RF=17AB8  000...2C8  NOMNEM HEX

          RF=1AFB8  23C

$DISPLAY  ON DISPLAYFILE  ORL=L  GRS  FR6
```

Note: Two instances of ambiguity in parameters may occur. One of these is EBCD (the other is BCD) which, when preceded and followed by blanks, can represent either a format parameter or the hexadecimal address of a single core location. The ambiguity is resolved by interpreting both EBCD and BCD as format parameters; to display the single location EBCD one may use the block notation EBCD...EBCD.

COMMAND DESCRIPTION

Name: DUMP

Purpose: Cause a dump of the job to be given.

Prototype:

$DUMP [ON aFDname] [ORL=x] [par 1] [par 2]...

where

aFDname — is the name of the file or device on which the dump is to be written.

x — takes the character value "L" or "S" to request long (130 characters) or short (70 characters) output records.

par i — are format parameters selected from the following list:

HEX	hexadecimal conversion
MNEM	mnemonic conversion
EBCD	EBCDIC conversion
BCD	same as EBCD
DSPC	double spaced
DBLS	same as DSPC
SP2	same as DSPC

Usage: This command is used to cause the dumping of general registers, floating point registers, and core storage associated with the job.

Default cases for omitted parameters:

aFDname	sink
x	L for line printers, S for console typewriters and teletypes, S for files
par i	HEX (only)

Effect: The parameters are inspected to insure that the file, if specified, exists or that the device, if specified, is an

output device which is free. If so, the remainder of the parameters are inspected. None, any, or all of the command parameters may be specified. The only restriction on the order in which command parameters are given is that "ON aFDname" must appear first if it appears at all. ORL=L may be overridden by a subsequent ORL=S in the parameter list (and vice versa). If "MNEM" is among the format parameters the specification of "HEX" is implied and need not be given.

The parameters are used to construct the calling sequence for the subroutine SDUMP and control is transferred to SDUMP once to dump the general register (hex conversion) and floating point registers (hex conversion) and subsequently to dump successive storage blocks associated with the job. The format parameters in the command line govern the format of the core storage output information. For details concerning formats for core storage dumps see the writeup on SDUMP.

Examples:

```
$DUMP

$DUMP ON PTR2 MNEM DSPC

$DUMP ON DUMPFILE ORL=L EBCD

$DUMP HEX BCD SP2
```

COMMAND DESCRIPTION

Name: EMPTY

Purpose: to clean out a file without destroying it

Prototype: $EMPTY Fname

where- Fname is the name of the file to be emptied.

Usage: This command is used to clean out a file when none of the previous contents are wanted. Complaint will be made if a file by that name does not exist. Library files may be emptied only if the command is issued from the operator's console. If the parameter is correct, confirmation is requested exactly as in $DESTROY (q.v.) unless the file is a scratch file.

Effect: All current contents of Fname are discarded.

Examples:

```
$EMPTY   SV

$EMPTY   FILE2
```

MTS-230/END-0

12-1-67

COMMAND DESCRIPTION

Name: ENDFILE

Purpose: Provide an end-of-file indication in the source stream.

Prototype: $ENDFILE

[Note that there is no abbreviation allowed]

Usage: This card, if it appears in the source stream, provides a local end-of-file indication. It is used to stop Fortran from reading source cards, stop COPY or LIST coming from *SOURCE*, and stop object programs from reading source cards.

Examples: $ENDFILE

COMMAND DESCRIPTION

Name: ERRORDUMP

Purpose: Allow automatic dumps in batch mode.

Prototype: $ERRORDUMP

Usage: If this command is placed in a batch job and a subsequent program executed as a result of $RUN or $LOAD/$START terminates abnormally a dump of the registers and storage is given. See also MTS-230/SET

Effect: Automatic dumps are allowed. This has no effect in terminal usage.

Example: $ERR

MTS-230/GET-0

12-1-67

COMMAND DESCRIPTION

Name: GET

Purpose: To obtain a file, either private or library, as the current file.

Prototype:

```
$GET Fname
```

where- Fname is the name of the file to be the current file.

Usage: This command is used to establish an existing file as the current file. Complaint will be made if a file by that name does not exist or if the name given is a device name. Both library and private files may be obtained as the active file in this manner, but a password must be provided to allow changing of a library file. If the password has not been established for the current file by means of $PASSWORD, the first attempt to enter a line in the file will cause a request for the password to be forthcoming.

Effect: Fname becomes the current file.

Examples:

```
$GET QQSV

$GET FIT3STANZA8
```

COMMAND DESCRIPTION

Name: HEXADD,HEXSUB

Purpose: to add or subtract two hexadecimal numbers.

Prototype: $HEX {ADD} XXXXXXXX XXXXXXXX
{SUB}
Where XXXXXXXX represents either a hexadecimal number of one to eight digits or GRx where x is an integer between 0 and 15.

Usage: The hexadecimal numbers are entered with one or more intervening blanks as delimiters. If GRx is used for one or both of them the contents of the appropriate general register is used. For $HEXADD the results appear in the form :

SUM = XXXXXXXX.

For $HEXSUB the results appear in the form:

DIFF = XXXXXXX.

Overflows are ignored and negative results obtained from $HEXSUB are given with a minus sign preceding the absolute value of the difference.

Example: $HEXADD 1A2 2E81D

$HEXSUB 003BC8 1EA

$HEX GR3 3EF

COMMAND DESCRIPTION

Name: LIST

Purpose: To list, with line numbers, a file or section of a file.

Prototype:

```
                        {ON}
$LIST  [fromFDname] [ {,  } ] [toFDname]
```

where-

fromFDname is the name of the file or device which is to be listed.

toFDname is the name of the file or device on which the listing is to be done.

Usage: This command causes the lines of a file (or device) to be copied onto a device (or file) preceded by their line numbers. Normal usage is to list a file on a terminal or a printer. Complaint will be made if either fromFDname or toFDname specify a file that does not exist, or a device that is either not available or is the wrong type (is output device, or input device, respectively).

Default values for parameters that are omitted when the command is used are:

fromFDname current file

toFDname sink (initially terminal)

If fromFDname is omitted or follows toFDname, either "ON" or a comma must precede toFDname.

Effect: For the specified line number range (if any), lines are read sequentially from fromFDname until an EOF condition is read. For each line, the line number is converted to 12 characters BCD. This is put on the front of the line, and this extended line is written (according to the modifiers given in the command) on to toFDname.

Examples:

```
$LIST
$LIST  FIT3STANZA7,PTR1
$LIST  FILE(20,32)
$LIST  OFILY(117,117)
$LIST  ON  LISTFILE
$LIST  AFD  *SINK*
```

COMMAND DESCRIPTION

Name: LOAD

Purpose: To load a program but not to start execution of it, so that parts of it may be altered (if necessary) before starting.

Description: This command is identical to $RUN (q.v.) except that at the point where execution of the program would begin, control is returned to the user in command mode. The program can then be displayed and altered, and finally execution begun with the $START command.

&TS-230/NUM-1

12-1-67

COMMAND DESCRIPTION

Name: NUMBER

Purpose: Start automatic numbering of input lines from source to the active file.

Prototype:

```
$NUMBER  {[startingnumber] [,] [increment]}
         {CONTINUE                        }
```

where-

startingnumber is the number the automatic line numbering is to start with.

increment is the number that is to be added to a line number to get the next one.

Usage: This command starts automatic generation of line numbers. This is so data cards without numbers can be read into a file, and so new programs can be entered from consoles more easily.

Default cases for parameters:

```
startingnumber    1
     increment    1
```

If CONTINUE is specified, numbering resumes from where it left off at the last UNNUMBER command.

Both starting number and increment can take the usual three forms of a line number

```
linenumber

LAST

LAST±linenumber
```

In the last two cases, the value of LAST is the line number of the last line in the current active file. If the file is empty, the value of LAST is zero.

Examples:

```
$NUMBER
$NUMBER  10,10
$NUMBER  ,.1
$NUMBER  CONTINUE
$NUM     1  .1
```

COMMAND DESCRIPTION

Name: PASSWORD

Purpose: Allow a library file to be changed.

Prototype: $PASSWORD [IS] password

where password is the required password.

Usage: This command sets a write-allow switch for library files. When the current file is changed, this switch is reset in order to protect the user from inadvertent modification of succeeding files. If the password given was correct, the response "O.K." is given. If the password was wrong, a negative response is given.

Effect: Write-allow switch is set for library files.

Examples: None, obviously.

Note: This is not the user-password. See sections describing $SIGNON and $SET for description of that facility.

COMMAND DESCRIPTION

Name: RESTART

Purpose: To restart a program after an interrupt.

Prototype:

```
$RESTART [[AT] [{RF=HHHH}] [HHHH]] [limit keyword pars]
         |     {RF=GRx }       |
         L                     J
```

The address to which control is to be given is determined in the following manner:

1. If no parameters are given the program is restarted with the PSW typed out at the time of the last interrupt.

2. If parameters are given the restart address is found by adding the relocation factor (default zero unless changed by a $SET command) to the displacement provided. The relocation factor must be given either as a free-form hexadecimal number or GRx in which case the contents of the specified general register is used. The displacement value must be given in free-form hexadecimal format.

Limit keyword parameters are described in the section on Limits Specification, section MTS-225.

Effect: The program is restarted at the address determined as described above. Complaint is made if the relocation factor and displacement are not legal or if the relocation factor is greater than the maximum core address. By appropriate use of an eight digit displacement, one can set the condition code or program mask when restarting a program; that is, the sum of the relocation factor and displacement is taken as the entire right hand half of the PSW.

Examples:

```
$RESTART

$RES  AT RF=GR15  0

$RESTART  AT  20A58

$RES    RF=20800  258
```

```
$RES     RF=20800      28000258
```

(In the last example the program is restarted at 20A58 with the condition code set to 2, the fixed point overflow interruption enabled, and the other program mask interrupts disabled.)

COMMAND DESCRIPTION

Name: RUN

Purpose: To load and initiate execution of a user's program

Prototype:

```
                                {mapFDname}
   $RUN  [[objectFDname] [ {NOMAP     } ]]  ; [logicalI/Ounit=aFDname ]...

          [limit keyword parameters]   [PAR=parameters]
```

where

objectFDname is the name of the file or device where the object cards are to be found.

mapFDname is the name of the file or device on which the loader will write the map. If NOMAP is specified, printing of the map will be suppressed; error comments (if any) will be printed on the user's terminal.

logicalIOunit the name of a "logical I/O unit" the object program has access to. Specifiable names are:

SCARDS
SPRINT
SERCOM
SPUNCH
numbers 0 to 9

aFDname The actual file or device on which or from which the data requested through the logicalIOunit is to go or come.

limit keyword parameters
These are time, cards, and page limits. They are described in the section Limits Specification, section MTS-225.

parameters Parameters to be passed to the program executed. Everything from the first character after the "=" to the end of the command is passed as parameters.

Usage:

The RUN command directs that an object deck is to be loaded and then run. Certain parameters must be specified for loading and certain parameters for execution; these are separated in the command by the semi-colon: loading parameters to the left, execution parameters to the right. The semi-colon may have one or more blanks before and/or after it, but they are not necessary.

The loading parameters specify the input and output devices for the loader. The execution parameters make the connection between the logical IO units used by the user in his program and the physical devices or files to be used for them in this run.

Default specifications, when parameters are not supplied, are:

objectFDname	source file or device
mapFDname	NOMAP
SCARDS	source
SPRINT	sink
SERCOM	terminal (master sink)
SPUNCH	none (if terminal usage) *PUNCH* (if batch)
0 to 9	none

LIBRARY FACILITY

If, after loading all that the RUN command has specified, there are still unresolved external symbol references, loading is continued from the system library, which is the file named *LIBRARY. This file is set up with the appropriate loader control cards so that only those pieces necessary are loaded. If there are still unresolved external symbol references after loading from this file, it is a fatal loading error.

Effect:

The parameters are inspected to make sure that the files specified exist and that the devices specified are free. If not, an error comment is produced and the logical I/O unit referring to the unavailable file or device is set up in such a way that the first time the program being run refers to it either the user is given a chance to respecify the name (if terminal usage) or the execution is terminated (if batch usage). The loader is called to load the object program into a region in core. If there were no fatal loading errors, control is transferred to the entry point of

the program by calling it as a subroutine via GR15 with return in GR14, parameter specification in GR1 (see below), and save area location in GR13 (standard OS CALL). The comment "EXECUTION BEGINS" is printed just before control is transferred. If the program terminates by restoring the registers and returning via GR14 (OS standard), the comment "EXECUTION TERMINATED" is printed and the command is then terminated by releasing all storage, files, and devices used. If execution is terminated for any other reason (for example the program calls ERROR) the storage, files, and devices are not released until the next $RUN or $LOAD command. This enables the terminal user to user to use $DISPLAY, $ALTER, and $RESTART to debug the program. An exception to this rule is if the first or only name in the object FDname concatenation is a library file, storage, files, and devices are always released at the end of execution. If the program gets a program interrupt, the comment:

PROGRAM INTERRUPT. PSW=XXXXXXXX XXXXXXXX

is printed, where the X's are the program old PSW that was stored because of the interrupt.

PARAMETER SPECIFICATION

The parameter (set up by the "PAR=" keyword specification) is passed as follows. Register 1 contains the location of a full-word adcon. This adcon is the location of a half-word count (halfword aligned) which is immediately followed by an EBCDIC character region (of the byte-length specified in the count) which contains the parameters. The left-most bit of the adcon will be 1. (This is standard OS setup.)

Examples:

```
$RUN

$RUN  -FYLE4 *SINK*

$RUN  *FORTRAN

$RUN *ASMBLR NOMAP; SCARDS=SFILE SPRINT=LISTING
          SPUNCH=SFILE
```

COMMAND DESCRIPTION

Name: SET

Purpose: Set various global switches and quantities.

Prototype: $SET kywd=quan

where: kywd may be

CASE — quan may be UC or LC. (LC is normal) If UC is in effect, all data lines read by the MTS monitor (See MTS-220 and MTS-241) will have lower case letters converted to upper case. Useful for terminals like 2741 and 1050 which have both cases, to avoid having to use the shift key so much.

RF — quan is a hexadecimal number or GRx where x is an integer between 0 and 15. This sets a global relocation factor quantity which is used in DISPLAY and ALTER commands. If GRx is specified the relocation factor is set to the contents of the specified general register. (value is zero initially)

IC — quan may be ON or OFF (is ON initially). If IC is on implicit concatenation is active as described in MTS-210. If IC is off no check is made for "$CONTINUE WITH" lines and they are treated as any other lines.

PW — quan is any sequence of zero to six characters none of which are blank. If at least one character is given the character string becomes the password for the current user and must be given correctly before someone is allowed to signon with his user id. If zero characters are given the password is no longer required. See description of $SIGNON for how to specify the password when signing on.

ERRORDUMP — quan may be ON or OFF (is OFF initially). If errordump is on and an execution in batch mode terminates

abnormally a dump is given. This has no effect in terminal usage.

LNS — quan is one character specifying the Line Number Separator (is a comma initially). The LNS is that character which, if it terminates the line number at the beginning of an input line, is not considered as part of the line but only as a separator. Hence a line commencing with numeric information may be easily entered. E.g., a line beginning: 174,10LINE = 2

LFR — quan may be ON or OFF (is ON initially). If ON, storage occupied by library files during a RUN is always released when they return to the system, no matter what the reason. If OFF, library files are treated the same as any other file in this respect. (See also $RUN description)

Examples:

```
$SET  CASE=UC

$SET  RF=2B800
```

COMMAND DESCRIPTION

Name:	SIGNOFF
Purpose:	Notify the system of the user's departure
Prototype:	$SIGNOFF
Usage:	Obvious
Effect:	All devices attached (and storage acquired) are released, all files are closed, and the system quiescently awaits the arrival of the next user on that terminal.
Example:	$SIGNOFF
Note:	If the abbreviation $SIG is used, its meaning is taken in context: if no one is signed on, $SIG means $SIGNON; if someone is signed on, $SIG means $SIGNOFF.

COMMAND DESCRIPTION

Name: SIGNON

Purpose: Identify the user to the system

Prototype: $SIGNON id [PW=cccc] [limit keyword parameters]

where id is the one to four character long user identification.

cccc is the password for id, if one is required.

limit keyword parameters specify limits on time, pages, and cards. They are described in the section Limit Specification, section MTS-225.

Usage: This must be the first command given. Complaint will be made if id is omitted. If more than four characters are given for id, only the first four will be used. See description of $SET command for setting of user passwords.

Effect: Four characters are stored to use as the user prefix for accessing files. If only one, two, or three characters are given as the I.D., this is padded out to four characters by adding .$., $., or ., respectively, to the characters given. If the I.D. given has a password and either none was given, or an incorrect one was given the user will be given a chance to enter the correct one (terminal usage) or will be thrown off (batch usage).

Example: $SIGNON Y025

Note: If the abbreviation $SIG is used, its meaning is taken in context: if no one is signed on, $SIG means $SIGNON; if someone is signed on, $SIG means $SIGNOFF.

Name: SINK

Purpose: Changes the sink for "normal output".

Prototype: $SINK {FDname / PREVIOUS}

where- FDname is the file or device name of the new sink.

Usage: Appearance of this command in the source stream causes the next output directed to the sink device to be put on file or device specified. The terminal the user is at always remains connected as the master sink on which error messages that require user action are given.

A one level pushdown of sink devices is maintained. If "PREVIOUS" is given instead of a file or device name, the previous sink device is restored.

This command is not allowed in batch operation.

Effect: File or device specified replaces sink. Msink remains unchanged.

Examples: $SINK PTR1

$SINK PRINTSYSOUT

$SINK PREVIOUS

MTS-230/SOU-0

12-1-67

COMMAND DESCRIPTION

Name: SOURCE

Purpose: Changes the source for command and data lines.

Prototype: $SOURCE {FDname / PREVIOUS}

where- FDname is the file or device name of the new source.

Usage: Appearance of this command in the source stream causes the next input line to be taken from the file or device specified. The terminal the user is at always remains connected as the master source from which attention (or similar) interrupts occur and on which error messages that require user action are given.

A one level pushdown of source devices is maintained. If "PREVIOUS" is given instead of a file or device name, the previous source device is restored.

This command is not allowed in batch operation.

Effect: File or device specified replaces source. Msource remains unchanged.

Examples:

```
$SOURCE  RDR1

$SOURCE  *DOJOB

$SOURCE  PREVIOUS
```

COMMAND DESCRIPTION

Name: START

Purpose: To start execution of a program after loading it (via $LOAD).

Description: This command is identical to $RESTART, which see.

COMMAND DESCRIPTION

Name: UNNUMBER

Purpose: Turn off automatic numbering of input lines.

Prototype: $UNNUMBER

Usage: This command is used when it is desired to suspend the automatic numbering of input lines that was turned on by use of the $NUMBER command.

Examples: $UNNUMBER

DATA LINES

LINE NUMBERS

All lines read by the MTS monitor that are not commands must start with a line number (except when the automatic numbering mode is on). In its full form, this line number contains a sign, five digits, a decimal point, and three digits. The sign need not be specified; if missing it is assumed positive. The decimal point and following fractional digits need not be specified; if missing, the number is assumed an integer. Leading zeros need not be specified in the integer part; trailing zeros need not be specified in the fraction part. Complaint will be made if more than five digits precede the decimal point or more than three digits follow it.

Examples of line numbers are:

5 5.1 5.13 5.137 32505.137 -32505.137

The first character following the line number is the first character of the line. The end of the line number is determined as follows:

1. An alphabetic character terminates the number.
2. The second occurrence of . terminates the number (first occurrence is the decimal point)
3. A + or - which is not the first character terminates the number.
4. A blank terminates the line number.
5. Any other special character terminates the line number.

If the character which terminates the line number is a Line Number Separator character (usually a comma), then this character is considered only a separator and is not taken as either part of the line number or as part of the line. For example, to put the three characters "123" in as line one of a file,

1,123

would be typed. (See the $SET command for changing this character.)

Two other forms of line numbers are permitted. They are

LAST
LAST±linenumber

The value of LAST is the line number of the last line in the current active file or device. Note that LAST-1 does not necessarily specify the line number of the next-to-last line; it is merely a line number 1 less than that of the last line.

USER PROGRAMS

This general section gives rules for writing programs so they can be run under MTS. Section 251 lists general constraints, section 252 describes how parameters and modifiers may be passed to the program, section 253 contains the subroutine descriptions for all of the subroutines available to the user's program, and section 255 contains description of the macro library available for the assembler.

USER PROGRAM CONSTRAINTS

Programs run under MTS are subject to the rules and constraints listed below:

1. The program runs in problem state.

2. Storage protection (both store and fetch protect) is on, with each user's program having a different key.

3. Programs are started up with an OS CALL: parameter list in location GR1, save area location in GR13, return address in GR14, entry point location in GR15.

4. Programs are expected to start by saving registers in the save area, and when they wish to return, they are expected to restore all registers and then branch to where GR14 points. Alternatively, the subroutines SYSTEM and ERROR are provided as means of exits.

5. All I/O shall be done by calling on subroutines provided.

6. No SVC instructions should be used.

I/O ROUTINES - PARAMETER DESCRIPTION

All the I/O routines have the same types of parameters, and in the same order. The exact calling sequences are given in later sections. The calling sequence is designed in such a way that these routines may be called from either assembly language or FORTRAN programs. The parameter descriptions, being common, are given here:

1. REGION — location of the core region to or from which data is to be transmitted.

2. COUNT — location of a half word integer which is the number of bytes to be transmitted.

3. MODIFIERS — location of a full word of switches used to modify action of the subroutine. Each switch is represented by two bits: an ON bit and an OFF bit. If either the ON bit or the OFF bit, but not both, is set, the modifier is or is not used, respectively. If neither or both of the bits are set, but the given modifier was applied to the FDname (See MTS-210), then what was specified by the FDname is used. If neither the subroutine call nor the FDname specifies which to use, a default specification is used. The modifiers and their default specifications are:

Bit number		Name	Default
31	OFF	indexed	OFF
30	ON		
29	OFF	binary	OFF
28	ON		
27	OFF	case conversion	OFF
26	ON		
25	OFF	carriage control and stacker select	ON for printers and terminals, OFF for all else
24	ON		

23	OFF		
		prefix print	OFF
22	ON		
21	OFF	peel	
		or	OFF
20	ON	return line nr	
19	OFF		
		machine	OFF
18	ON	carriage control	
17	OFF		
		trim	ON
16	ON		
15	OFF		
		special	OFF
14	ON		

Explanation of the modifiers is later in this section.

4. LINE NUMBER — location of a full word which contains the line number transmitted to or returned from the subroutine. The line number is stored internally as a full word integer whose value is the line number times 1000.

5. UNIT NUMBER — location of a full word which contains either the logical unit number as a full-word integer or the location of a FDUB (as returned by GETFD) as a full-word.

EXPLANATION OF MODIFIER BITS

Bits	Name	
31}	SEQUENTIAL	Indexed and Sequential are opposites.
30}	INDEXED	Indexed means the user is specifying the line number; i.e., he says "put (or get) this line at (from) this place". Sequential means that the user wants to get (or put) the next line.
29}	EBCD	Normal mode is BCD. The binary modifier is
28}	BINARY	used to read or punch binary cards. It has nothing to do with binary tape.
27}	LC	Normally characters are transmitted
26}	UC	unchanged. If this modifier is on, lower case letters are changed to upper case letters.
25}	CC	Controls carriage control or stacker select
24}	SS	(which is which depends on the device) Normal carriage control is "logical carriage control". Legal carriage controls are:

blank	single space
-	triple space
0	double space
1	start new page
9	single space and suppress overflow

If stacker select for punch is specified, first character is taken as control:

0	for stacker	1
1		2
2		3

If first character is a legal control it is so used, and characters 2 and on are punched in column 1 and on; if not legal, the card is punched as is (character 1 and on into column 1 and on) and stacked in stacker 2.

For control functions for tapes, see Tape User's Guide (MTS-160).

23}	PREFIX	If on, for both input and output, the current
22}		line number is converted to external form and printed as a prefix on the appropriate input or output line.

21} 20}	PEEL	input If on, a line number is "peeled" off the front of the line. The number is converted to internal form and returned; the rest of the line is returned as the input line. output If on, forces the line number in the sequential case to be returned to the spot designated in the parameter list. (not normally the case).
19} 18}	MCC	If on, then machine carriage control is used. In this case, for printers (and simulated for terminals) , the first byte of the line is used as the command in the CCW used for output, if legal, and the first byte is blanked out. If not legal, the entire line is printed single spaced. For all other devices and for files, this modifier is ignored.
17} 16}	TRIM	Normally TRIM is on. This means that if a line has any trailing blanks, all but one are deleted.
15} 14}	SPECIAL	This modifier is reserved for device dependent uses. Its meaning depends on the particular device type it is used with.

SUBROUTINE DESCRIPTIONS

This section lists those subroutines available to the user. Most of them are in the system library (file *LIBRARY), which is automatically searched when a program is run via the $RUN command (and also when the LOAD, LINK, and XCTL subroutines are called).

In addition to those described in this section, all of the subroutines described in "IBM SYSTEM/360 OPERATING SYSTEM: FORTRAN IV Library Subprograms",form C28-6596, are in the system library.

IBM's Scientific Subroutine Package (SSP), Version II, described in form H20-0205, is in the file *SSP. If usage of any of the subroutines in SSP occurs, this file should be concatenated onto the object file(s) loaded when the program is run, e.g:

```
$RUN OBJECT+*SSP *SINK*;5=*SOURCE* 6=*SINK*
```

MTS System and Library External Symbols

Following are the lists of all available external symbols. The corresponding subroutine descriptions are in this section (MTS-253) and IBM Publications: System/360 Scientific Subroutine Package (360A-CM-03X) Version II, Programmer's Manual (H20-0205) and System/360 Operating System: FORTRAN IV Library Subprograms (C28-6596). See the writeup on *SSP (MTS-280/62475) for details on referring to SSP entry points.

Pre-Defined External Symbols

ERROR	LCSYMBOL	READ	SCARDS	SDUMP	SERCOM	SPRINT
SPUNCH	SYSTEM	WRITE				

Low-Core External Symbols

CANREPLY	EMPTY	ERR	ERRCOM	FREEFD	FREESPAC	GDINFO
GETFD	GETSPACE	GUSERID	IN	INPUT	LINK	LOAD
MAKFFILE	OUT	OUTPUT	PGNTTRP	PRINT	PUNCH	SETIOERR
SETPFX	UNLOAD	XCTL				

Library Entry Points and Module Names

#FPCON	@TESTITP	ACCEPT	ADCON#	ALGAMA	ALMADGO	ALMADP
ALMADR	ALMADW	ALOG	ALOG10	AMAX0	AMAX1	AMIN0
AMIN1	AND	ARCOS	ARSIN	ATAN	ATAN2	CABS
CCOS	CDABS	CDCOS	CDDVD#	CDEXP	CDLG10	CDLOG
CDMPY#	CDSIN	CDSQRT	CDVD#	CEXP	CLOG	CLOG10
CMPY#	COMPL	COS	COSH	COTAN	CSIN	CSQRT
DARCOS	DARSIN	DATAN	DATAN2	DCOS	DCOSH	DCOTAN
DEBUG#	DERF	DERFC	DEXP	DGAMMA	DIOCS#	DISMOUNT
DLGAMA	DLOG	DLOG10	DMAX1	DMIN1	DROPIOER	DSIN
DSINH	DSQRT	DTAN	DTANH	DUMP	DVCHK	D7090
D7090P	ERF	ERFC	EXIT	EXP	E7090	E7090P
FCDXI#	FCVA0	FCVC0	FCVE0	FCVI0	FCVL0	FCVTHB
FCVZ0	FCXPI#	FDIOCS#	FDXPD#	FDXPI#	FIOCS#	FIXPI#
FPAUS#	FPC	FRDNL#	FRXPI#	FRXPR#	FSTOP#	FWRNL#
GAMMA	GLAP	IBCOM#	IBERH#	IBEXIT#	IBFERR#	IBFINT#
IHCCLABS	IHCCLAS	IHCCLEXP	IHCCLLOG	IHCCLSCN	IHCCLSQT	IHCCSABS
IHCCSAS	IHCCSEXP	IHCCSLOG	IHCCSSCN	IHCCSSQT	IHCDBUG	IHCDIOSE
IHCFCDXI	IHCFCOMH	IHCFCOMM	IHCFCVTH	IHCFCXPI	IHCFDUMP	IHCFDVCH
IHCFDXPD	IHCFDXPI	IHCFEXIT	IHCFIOSH	IHCFIXPI	IHCFMAXD	IHCFMAXI
IHCFMAXR	IHCFOVER	IHCFRXPI	IHCFRXPR	IHCFSLIT	IHCIBERH	IHCLASCN
IHCLATN2	IHCLERF	IHCLEXP	IHCLGAMA	IHCLLOG	IHCLOGIC	IHCLSCN
IHCLSCNH	IHCLSQRT	IHCLTANH	IHCLTNCT	IHCNAMEL	IHCSASCN	IHCSATN2
IHCSERF	IHCSEXP	IHCSGAMA	IHCSLOG	IHCSSCN	IHCSSCNH	IHCSSQRT

IHCSTANH	IHCSTNCT	IHCUATBL	IOHERP	IOHERR	IOHETC	IOHIN
IOHLINK	IOHOUT	IOHSFSET	IOH360	IOPKG	IOPMOD	LAND
LCLOSE	LCOMPL	LOPEN	LOR	LXOR	MAXO	MAX1
MINO	MIN1	MOUNT	OMIT	ONE@ATIM	OR	OVERFL
OWNCONVR	PCCLOSE	PCLOSE	PCOPEN	PDUMP	PLOT	PLOTP
PLOT1	PLOT14	PLOT2	PLOT3	PLOT4	POPEN	QCLOSE
QCNTRL	QDTAN	QGET	QGETUCB	QGTUCB	QOPEN	QPUT
QSAM	QSAMP	QTAN	RCLOSE	REWIND	REWIND#	ROPEN
SERCLOSE	SEROPEN	SETEOF	SETERR	SETFRVAR	SETIOHER	SETLOG
SHFTL	SHFTR	SIN	SINH	SLITE	SLITET	SPIE
SQRT	STPLT1	STPLT2	SYSC	SYSL	SYSP	SYSR
TAN	TANH	XOR				

SSP Entry Points and Module Names

ABSNT	ABSNT#	ACFI	ACFI#	AHI	AHI#	ALI
ALI#	ARRAY	ARRAY#	ATSE	ATSE#	ATSG	ATSG#
ATSM	ATSM#	AUTO	AUTO#	AVCAL	AVCAL#	AVDAT
AVDAT#	BESI	BESI#	BESJ	BESJ#	BESK	BESK#
BESY	BESY#	BOUND	BOUND#	CADD	CADD#	CANOR
CANOR#	CCPY	CCPY#	CCUT	CCUT#	CEL1	CEL1#
CEL2	CEL2#	CHISQ	CHISQ#	CINT	CINT#	CNP
CNP#	CNPS	CNPS#	CONVT	CONVT#	CORRE	CORRE#
CROSS	CROSS#	CS	CS#	CSP	CSP#	CSPS
CSPS#	CSRT	CSRT#	CSUM	CSUM#	CTAB	CTAB#
CTIE	CTIE#	DACFI	DACFI#	DAHI	DAHI#	DALI
DALI#	DATSE	DATSE#	DATSG	DATSG#	DATSM	DATSM#
DCEL1	DCEL1#	DCEL2	DCEL2#	DCLA	DCLA#	DCNP
DCNP#	DCNPS	DCNPS#	DCPY	DCPY#	DCSP	DCSP#
DCSPS	DCSPS#	DELI1	DELI1#	DELI2	DELI2#	DFMCG
DFMCG#	DFMFP	DFMFP#	DGELB	DGELB#	DGELG	DGELG#
DGELS	DGELS#	DHARM	DHARM#	DHEP	DHEP#	DHEPS
DHEPS#	DHPCG	DHPCG#	DHPCL	DHPCL#	DISCR	DISCR#
DJELF	DJELF#	DLAP	DLAP#	DLAPS	DLAPS#	DLBVP
DLBVP#	DLEP	DLEP#	DLEPS	DLEPS#	DLLSQ	DLLSQ#
DMATX	DMATX#	DMFGR	DMFGR#	DPECN	DPECN#	DPECS
DPECS#	DPRQD	DPRQD#	DQATR	DQATR#	DQA12	DQA12#
DQA16	DQA16#	DQA24	DQA24#	DQA32	DQA32#	DQA4
DQA4#	DQA8	DQA8#	DQG12	DQG12#	DQG16	DQG16#
DQG24	DQG24#	DQG32	DQG32#	DQG4	DQG4#	DQG8
DQG8#	DQHFE	DQHFE#	DQHFG	DQHFG#	DQHSE	DQHSE#
DQHSG	DQHSG#	DQH16	DQH16#	DQH24	DQH24#	DQH32
DQH32#	DQH48	DQH48#	DQH64	DQH64#	DQH8	DQH8#
DQL12	DQL12#	DQL16	DQL16#	DQL24	DQL24#	DQL32
DQL32#	DQL4	DQL4#	DQL8	DQL8#	DQSF	DQSF#
DQTFE	DQTFE#	DQTFG	DQTFG#	DRHARM	DRHARM#	DRKGS
DRKGS#	DRTMI	DRTMI#	DRTNI	DRTNI#	DRTWI	DRTWI#
DTCNP	DTCNP#	DTCSP	DTCSP#	DTEAS	DTEAS#	DTEUL
DTEUL#	DTHEP	DTHEP#	DTLAP	DTLAP#	DTLEP	DTLEP#
EIGEN	EIGEN#	ELI1	ELI1#	ELI2	ELI2#	EXPI
EXPI#	EXSMO	EXSMO#	FMCG	FMCG#	FMFP	FMFP#
FORIF	FORIF#	FORIT	FORIT#	GAMMA	GAMMA#	GAUSS

GAUSS#	GDATA	GDATA#	GELB	GELB#	GELG	GELG#
GELS	GELS#	GMADD	GMADD#	GMPRD	GMPRD#	GMSUB
GMSUB#	GMTRA	GMTRA#	GTPRD	GTPRD#	HARM	HARM#
HEP	HEP#	HEPS	HEPS#	HPCG	HPCG#	HPCL
HPCL#	JELF	JELF#	KRANK	KRANK#	LAP	LAP#
LAPS	LAPS#	LBVP	LBVP#	LEP	LEP#	LEPS
LEPS#	LLSQ	LLSQ#	LOAD	LOAD#	LOC	LOC#
MADD	MADD#	MATA	MATA#	MCPY	MCPY#	MEANQ
MEANQ#	MFGR	MFGR#	MFUN	MFUN#	MINV	MINV#
MOMEN	MOMEN#	MPRD	MPRD#	MSTR	MSTR#	MSUB
MSUB#	MTRA	MTRA#	MULTR	MULTR#	NROOT	NROOT#
ORDER	ORDER#	PADD	PADD#	PADDM	PADDM#	PCLA
PCLA#	PCLD	PCLD#	PDER	PDER#	PDIV	PDIV#
PECN	PECN#	PECS	PECS#	PGCD	PGCD#	PILD
PILD#	PINT	PINT#	PMPY	PMPY#	PNORM	PNORM#
POLRT	POLRT#	PQSD	PQSD#	PRQD	PRQD#	PSUB
PSUB#	PVAL	PVAL#	PVSUB	PVSUB#	QATR	QATR#
QA10	QA10#	QA2	QA2#	QA3	QA3#	QA4
QA4#	QA5	QA5#	QA6	QA6#	QA7	QA7#
QA8	QA8#	QA9	QA9#	QG10	QG10#	QG2
QG2#	QG3	QG3#	QG4	QG4#	QG5	QG5#
QG6	QG6#	QG7	QG7#	QG8	QG8#	QG9
QG9#	QHFE	QHFE#	QHFG	QHFG#	QHSE	QHSE#
QHSG	QHSG#	QH10	QH10#	QH2	QH2#	QH3
QH3#	QH4	QH4#	QH5	QH5#	QH6	QH6#
QH7	QH7#	QH8	QH8#	QH9	QH9#	QL10
QL10#	QL2	QL2#	QL3	QL3#	QL4	QL4#
QL5	QL5#	QL6	QL6#	QL7	QL7#	QL8
QL8#	QL9	QL9#	QSF	QSF#	QTEST	QTEST#
QTFE	QTFE#	QTFG	QTFG#	RADD	RADD#	RANDU
RANDU#	RANK	RANK#	RCPY	RCPY#	RCUT	RCUT#
RECP	RECP#	RHARM	RHARM#	RINT	RINT#	RKGS
RKGS#	RK1	RK1#	RK2	RK2#	RSRT	RSRT#
RSUM	RSUM#	RTAB	RTAB#	RTIE	RTIE#	RTMI
RTMI#	RTNI	RTNI#	RTWI	RTWI#	SADD	SADD#
SCLA	SCLA#	SCMA	SCMA#	SDIV	SDIV#	SICI
SICI#	SIMQ	SIMQ#	SMO	SMO#	SMPY	SMPY#
SRANK	SRANK#	SRMA	SRMA#	SSUB	SSUB#	SUBMX
SUBMX#	SUBST	SUBST#	TAB1	TAB1#	TAB2	TAB2#
TALLY	TALLY#	TCNP	TCNP#	TCSP	TCSP#	TEAS
TEAS#	TEUL	TEUL#	THEP	THEP#	TIE	TIE#
TLAP	TLAP#	TLEP	TLEP#	TPRD	TPRD#	TRACE
TRACE#	TTEST	TTEST#	TWOAV	TWOAV#	UTEST	UTEST#
VARMX	VARMX#	WTEST	WTEST#	XCPY	XCPY#	

SUBROUTINE DESCRIPTION

Name: ATTNTRP

Purpose: To set things up so control will be returned to the user on an attention interrupt (ATTN key on 2741, BREAK on teletype, etc.).

Calling Sequence:

OS (I) R type
Register 13 save area is not required.

Entry:

GR0 should contain zero or the location to transfer control to if an attention interrupt occurs.

GR1 contains the location of a 72 byte save region in which are stored pertinent information (see below).

Description:

A call on this routine sets up an attention interrupt exit for one interrupt only. When an interrupt occurs and the exit is taken, it is reset so that another call on this subroutine is necessary to intercept the next attention interrupt.

When the attention interrupt exit is taken, the second four bytes of the save region contains the interrupt location, and the rest of it contains the general purpose registers in order 0 through 15. The floating point registers remain as they were at the time of the interrupt. GR1 contains the location of the save region.

If, on a call to this routine, the first byte of the region specified by GR1 is X'FF', then in addition to setting up the attention interrupt exit, the program is restarted at the location specified in the region (using that four bytes as the right-hand half of the PSW), using the GPR's in the region.

SUBROUTINE DESCRIPTION

Bitwise Logical Functions

For Fortran IV Users

AND,LAND,OR,LOR,XOR,LXOR,COMPL,LCOMP,SHFTR,SHFTL

These simple functions do the bitwise logical operations which are difficult to state in FORTRAN IV arithmetic formulas. If their names are prefixed with an "L", they are integer; otherwise they are declared real. The only exception to this rule is that SHFTR and SHFTL must be declared integer.

Unless otherwise stated, the arguments of the functions may be either real or integer provided that they are full words, i.e., four bytes long.

```
 AND        C = AND(A,B)
LAND       IC = LAND(IA,IB)
```

The result has bits on only if the corresponding bits of the arguments are both on.

```
 OR         C = OR(A,B)
LOR        IC = LOR(IA,IB)
```

The result has bits on only if either argument or both has the corresponding bits on.

```
 XOR        C = XOR(A,B)
LXOR       IC = LXOR(IA,IB)
```

The result has bits on only if the corresponding bits of the two arguments are not the same.

```
 COMPL      B = COMPL(A)
LCOMPL     IB = LCOMPL(IA)
```

The result simply has bits of the argument all reversed.

```
SHFTL      IC = SHFTL(IA,IB)
SHFTR      IC = SHFTR(IA,IB)
```

As logical shift functions they are not equivalent to a division or a multiplication by a power of two. The first argument is shifted right or left the number of bits specified by the second integer argument.

Examples:

Zero out all bits or a full word.

```
WORD = XOR(WORD, WORD)
```

Examine the second byte of a full word.

```
 DATA MASK/Z00FF0000/
SCDBYT = AND(WORD,MASK)
```

Move the first byte of a fullword so that it becomes the fourth byte of the word.

```
IWORD = SHFTR(IWORD,24)
```

Pack the four characters together.

```
      READ (5,4) (CHAR(I),I=1,4)
4        FORMAT (4A1)
      DATA MASK/ZFF000000/
      WORD = 0.
      DO 6 I=1,4
      WORD = OR(WORD,SHFTR(CHAR(I),(I-1)*8)
6       CONTINUE
```

SUBROUTINE DESCRIPTION

Name: Blocked Input/Output Routines

The blocked input/output routines have the following entry points:

QGETUCB, QOPEN, QCLOSE, QGET, QPUT, QCNTRL

Purpose: These routines will read and write blocked records consisting of one or more fixed-length logical records. All input/output requests are made for logical records; the routine handles record blocking and deblocking automatically. These routines are intended for use with tape records although they are not restricted to tapes. More than one input/output file or device may be handled at one time. The type of processing done by these routines is similar to that done by the Queued Sequential Access Method (QSAM) within OS and for this reason are sometimes referred to as the MTS QSAM routines. They should not however, be confused with the OS routines of the same name because the MTS routines provide only a subset of the features of the OS routines.

Several error messages can be generated. Each of these begins with the prefix:

QSAM ERROR: <FDname>

which will be abbreviated as "•••" in what follows.

The error messages which can be generated by each routine will be listed with that routine in the subroutine descriptions which follow:

BLOCKED I/O ROUTINE: QGETUCB (QGTUCB)

Name: QGETUCB (QGTUCB)

Purpose: To acquire a file or device which will be used by the blocked input/output routines and generate a table of control information for that file or device. This table is referred to as a UCB (Unit Control Block).

Calling sequence: OS (I) S type

Entry: GR1 contains the location of a parameter list:

Word 1: Location of the name of the file or device which is to used by the blocked input/output routines, ending with a blank or comma. The name may not be longer than sixteen characters.

Word 2: Address of a word in which the pointer to the UCB will be placed.

Return:

RC=0 Successful return. File or device was acquired and can now be used by the other blocked input/output routines.

RC=4 File or device could not be acquired properly from MTS. The subroutine GETFD or GDINFO returned a non-zero return code.

Messages: ••• COULD NOT BE ACQUIRED FROM MTS.

Description: A chain of all UCB'S acquired thus far is searched to see if this file or device has been set up before. If so, the UCB pointer is returned immediately. Otherwise, a UCB is built and added to the chain, a pointer to it is returned, GETFD and GDINFO are called for the file or device, and pertinent information is set up in the UCB.

BLOCKED I/O ROUTINE: QOPEN

Name: QOPEN

Purpose: To prepare a file or device which has been acquired by QGETUCB for blocked input or blocked output operations.

Calling sequence: OS (I) S type.

Entry: GR1 contains the location of a parameter list:

Word 1: Address of a word containing a UCB pointer as returned by QGETUCB.

Word 2: Address of a full-word integer which indicates whether information is to be read or written:
1=Information is to be written.
2=Information is to be read.

Word 3: Address of the full-word integer maximum number of logical records per physical block.

Word 4: Address of the full-word integer length of each logical record (in bytes). Legal values range from 2 through 256 bytes.

Return:

RC=0 Successful return. File or device can now be read from via QGET (if the second parameter value is 2) or written via QPUT (if the second parameter value is 1).

RC=4 The file or device is already open, or the second parameter is not a 1 or 2, or the maximum length of a physical record for tape was rejected by the tape device support routines (it probably was longer than 32767 bytes).

Messages:

••• IS ALREADY OPEN. IT CAN'T BE OPENED TWICE.
••• READ/WRITE SPECIFICATION INCORRECT IN CALL TO OPEN.
••• MAXIMUM RECORD LENGTH REJECTED BY TAPE DSR.

Description: The parameters are checked for consistency. The information from the parameters is placed in the UCB. The largest possible physical record length is computed and a buffer of that length is acquired. If the device is a tape, the tape device support routines are initialized to accept records up to the maximum length.

BLOCKED I/O ROUTINE: QGET

Name: QGET

Purpose: To acquire the next logical record from a file or device which has been opened as an input file or device via QOPEN.

Calling sequence: OS (I) S type.

Entry: GR1 contains the location of a parameter list:

Word 1: Address of the area in which the next logical record will be stored. (input area)

Word 2: Address of a word containing a UCB pointer as returned by QGETUCB.

Return:

RC=0 Successful return. The next logical record has been placed in the input area.

RC=4 End-of-file. The input area is sprayed with the character having FF as its hexadecimal representation. This corresponds to the 12-11-0-7-8-9 punched card code.

ERROR A message is printed and the subroutine ERROR called if the file or device has not been opened for input via the subroutine QOPEN, if an end-of-file indication has already been returned once for this file or device, if a physical record is longer than the maximum size computed from the parameters to QOPEN, or an error indication was received while reading the next physical record.

Messages:

••• USED IN GET ALTHOUGH NOT OPENED AS AN INPUT FILE.
••• USED IN GET ALTHOUGH END-OF-FILE INDICATION GIVEN.
••• INPUT RECORD IS LONGER THAN MAXIMUM SPECIFIED.
••• RETURN CODE GREATER THAN 4 FROM READ IN GET.

Description: Physical records are read from the file or device as required. Each physical record is broken into one or more logical records of the length specified in the call to QOPEN. One logical record is returned upon each call upon QGET. The last logical record in a physical record may actually be shorter than the length of a logical record, in which case it is padded out with blanks. If there are no more logical records the input area is sprayed with the character having FF as its hexadecimal representation. All necessary indices are maintained in the UCB.

BLOCK I/O ROUTINE: QPUT

Name: QPUT

Purpose: To write the next logical record to a file or device which has been opened as an output file or device via QOPEN.

Calling sequence: OS (I) S type

Entry: GR1 contains the location of a parameter list:

Word 1: Address of the area in which the next logical record is stored. (output area)

Word 2: Address of a word containing a UCB pointer as returned by QGETUCB.

Return:

RC=0 Successful return. The next logical record has been placed into the current physical record.

ERROR A message is printed and the subroutine ERROR called if the file or device has not been opened for output via the subroutine QOPEN or if a non-zero return code was received from WRITE while writing out a physical record.

Messages:

••• USED QPUT ALTHOUGH NOT OPENED AS AN OUTPUT FILE.
••• APPEARS TO BE FULL. (RC>0 FROM WRITE)

Description: Each logical record presented by a call upon QPUT is placed into a buffer. When the buffer becomes full, it is written out as one physical record. All buffers will contain the maximum number of logical records specified in the call to QOPEN except the last buffer which will be truncated if it is only partially full when QCLOSE is called. All necessary indices are maintained in the UCB.

BLOCKED I/O ROUTINE: QCLOSE

Name: QCLOSE

Purpose: To terminate blocked input or output operations on a file or device which has been opened via QOPEN. If the file or device was used for output and a partial buffer of logical records for it is present, it is written out as a part of the closing procedure.

Calling sequence: OS (I) S type.

Entry: GR1 contains the location of a parameter list:

Word 1: Address of a word containing a UCB pointer as returned by QGETUCB for the file or device to be closed. The word should contain a zero if all the currently open files or devices are to be closed.

Return: RC=0 All returns are successful, even though some error messages may have been printed.

Messages:
••• APPEARS TO BE FULL. (RC>0 FROM WRITE)
••• FISHY RETURN FROM FREESPACE.

Description: If the file or device was used for output and a partial buffer of logical records for it is present, it is written out. All information in the UCB is reset to the normal state of an unopened file or device. The file or device is available for use and can be re-opened or positioned.

Note: No tape mark is written when an output file is closed. It is the user's responsibility to see that a tape mark is written on a tape before it is rewound to mark the end of the information written. The subroutine QCNTRL can be used for this purpose.

BLOCKED I/O ROUTINE: QCNTRL

Name: QCNTRL

Purpose: To position or write tape marks on a tape which has been acquired for use by the blocked input/output routines.

Calling sequence: OS (I) S type.

Entry: GR1 contains the location of a parameter list:

Word 1: Location of the three byte logical carriage control used to perform the function required. See section MTS-160.

Word 2: Address of a word which contains a UCB pointer as returned by QGETUCB.

Return:

RC=0 Successful return. Operation was accepted by the tape device support routines.

RC=4 The file or device is currently open, is not a tape, was not successfully acquired by QGETUCB, or the logical carriage control was rejected by the tape device support routines.

Messages:

••• CANNOT BE POSITIONED BECAUSE IT IS OPEN.
••• CANNOT BE POSITIONED BECAUSE IT IS NOT A TAPE.
••• DOES NOT HAVE A FDUB AND SO CAN'T BE POSITIONED.
••• RC>0 FROM "CARRIAGE CONTROL" TO WRITE.

SUBROUTINE DESCRIPTION

Name: CANREPLY

Purpose: Find out if the user is at a terminal or if this is a batch job.

Calling sequence: OS (I)

Exit:

RC=0	yes	(conversational)
4	no	(batch)

SUBROUTINE DESCRIPTION

Name: DISMOUNT

Purpose: Allow users to dismount private volumes.

Calling sequence: OS (I) S type

Entry: GR1 contains the location of a pointer to the parameter list. The parameter list consists of a halfword length of a character string immediately followed by that character string. The character string should be those characters which would immediately follow the "PAR=" if this subroutine were being run as a library file. See the description of the library file *DISMOUNT, MTS-280/24627, for details.

SUBROUTINE DESCRIPTION

Name: EMPTY

Purpose: Do the same thing as the command $EMPTY, that is, remove all the lines from a file without destroying it.

Calling sequence: OS (I) R type

Entry: GR0 contains the location of a FDUB (such as GETFD returns)

Exit:
RC=0 O.K.
=4 File does not exist.
=8 Something wrong with the file.

SUBROUTINE DESCRIPTION

Name: ERROR

Purpose: Terminate execution with an error indication

Calling sequence: None - just branch

Effect: Returns control to MTS to terminate execution. The comment "ERROR RETURN" is printed. In BATCH mode, a dump is automatically given, if $ERRORDUMP or $SET ERRORDUMP=ON was specified.

SUBROUTINE DESCRIPTION

Name: E7090,D7090,E7090P,D7090P

Purpose: To allow users to convert 7090 floating point internal representation to one of the two types of internal floating point representations available on the 360.

Calling Sequence: OS (I) S type

Entry: GR1 contains the location of a parameter list which consists of two adcons. The first adcon points to the area from which the input is to be taken (either twelve or six bytes in length depending on the entry used) and the second adcon points to the output region (either four or eight bytes depending on the entry used).

Description: D7090 and E7090 expect the input to be twelve bytes long --- the low order three bits of each byte are taken as one octal digit. The sign of the number is assumed to be the first bit of the first octal digit. D7090P and E7090P assume a six byte input area. The low-order six bits of each byte are taken as two octal digits. The first bit of the first octal digit is taken as the sign. D7090 and D7090P assume the output area is eight bytes long. E7090 and E7090P assume that the output area is four bytes long --- since the mantissa of single-word floating point numbers in the 360 contains only 24 bits and the mantissa in a 7090 floating point number contains 27 bits, rounding (if any) is done for the E-type conversions.

Exit: RC= 0 conversion was successful

RC= 4 parameter list was not full-word aligned

SUBROUTINE DESCRIPTION

Name: FCVTHB

Purpose: To alter the disposition of blanks with respect to FORTRAN formatted-input on numeric fields.

Calling Sequence (in FORTRAN): CALL FCVTHB(ARG)

If ARG is a full-word zero, all blanks are assumed to be zeros on numeric input (this is the normal FORTRAN mode). If ARG is non-zero, all blanks are ignored.

SUBROUTINE DESCRIPTION

Name: FREEFD

Purpose: Free a file or device acquired with the GETFD subroutine.

Calling sequence: OS (I) R type

Entry: GR0 contains FDUB location as returned by GETFD or GDINFO.

Exit: RC=0 successful return.

=4 the FDUB wasn't or something else was fishy.

SUBROUTINE DESCRIPTION

Name: FREESPAC

Purpose: To release storage that was obtained with GETSPACE

Calling sequence: OS (I) R type
Register 13 save area is not required.

Entry: GR0 = 0 entire region allocated is released.
≠ 0 is length of region to be released. If not a multiple of 8, next shortest multiple of 8 is used.

GR1 location of first byte of region.

Exit: RC = 0 ok

4 (a) space was not initially allocated by GETSPACE and cannot be released.

(b) region (LOC to LOC+LEN-1) specified is not completely within a region.

SUBROUTINE DESCRIPTION

Name: GDINFO

Purpose: Obtain information about a file or device.

Calling sequence: OS (I) R type

Entry: either (a) GR0 contains a FDUB location (such as GETFD returns), and GR1 contains zero;

or (b) GR0 and GR1 contain an 8 character logical I/O unit name, left-justified with trailing blanks. [See the description of $RUN for list of logical I/O unit names.]

Exit: RC=0 ok. GR1 contains the location of a full word aligned region of information. (If a concatenation was specified in the original logical I/O unit setup or GETFD call, the information returned in this region applies to the currently active member of the concatenation.) The region contains:

WORD 1: FDUB pointer

WORD 2: 4 character BCD type (Section MTS-210 has list of device types. For file the type returned is FILE. For a non-existent file the type is NONE.)

RC=4 Something wrong. If "(a) call", the FDUB pointer wasn't. If "(b) call", either the name given was not a legitimate logical I/O unit name or else there was no file or device assigned to that logical I/O unit name.

Note: The storage pointed to by GR1 was allocated by GETSPACE and the user may call FREESPAC (with GR0 = 0) to release it when it is no longer needed.

SUBROUTINE DESCRIPTION

Name: GETFD

Purpose: Obtain a file or device

Calling sequence: OS (I) R type

Entry: GR1 contains the location of the first character of the FDname of what is wanted. The name must be terminated by a blank. The name does not have to be aligned.

Exit: RC=0 successful return. GR0 contains the FDUB location.

=8 device is busy

=12 device is not operational

Effect: If a device, the device is acquired. If a file, the file is not opened until the first usage. Thus this subroutine cannot tell if the file exists or not. The caller can tell if the file exists by calling GDINFO. The name may be a concatenation of file or device names each followed by modifiers or a line number range as described in MTS-210. If the FDUB pointer returned is used in a call to READ or WRITE the the modifiers or line number ranges will be used and if a concatenation was specified, the usual sequencing through the concatenation will take place.

SUBROUTINE DESCRIPTION

Name: GETSPACE

Purpose: To obtain storage

Calling sequence: OS (I) R type

Register 13 save area is not required.

Entry: GR0 has switches

Bit 31 = 1 Return not made until space available.

0 Return always made with RC indicating whether space available.

Bit 30 = 1 Storage obtained is associated with the current level of LINK so that it is release at the next return from a LINK or the next XCTL.

0 Storage obtained is associated with highest level program so it is not release until execution is terminated.

Bit 29 = 1 Attach storage obtained to system level.
Other bits in GR0 must be zero.

GR1 has length desired. If not multiple of 8, next largest multiple will be used.

Exit: GR1 has location of first byte of region obtained.

RC = 0 all ok; region returned.

4 space not available.

SUBROUTINE DESCRIPTION

Name: GUSERID

Purpose: To obtain the current 4 character user id (from $SIGNON)

Calling sequence: OS (I) R type
register 13 save area is not required.

Exit: GR1 contains 4 character i.d.

SUBROUTINE DESCRIPTION

Name: IOPMOD

Purpose: To allow the user to set the modifier bits and line numbers of the parameter lists used by the system supplied IOH360 OPEN-CLOSE routines (ROPEN,RCLOSE,POPEN,PCLOSE,PCOPEN,PCCLOSE,LOPEN, LCLOSE,SEROPEN,SERCLOSE) when such routines call READ,WRITE,SCARDS,SPRINT,SPUNCH, or SERCOM.

Calling sequence: OS (I) S type

Entry: GR1 contains the location of a parameter list which consists of four adcons. The first adcon points to a full word which contains the new set of modifier bits for the logical I/O unit in question. The second adcon points to a full-word location containing either a logical unit number or a FDUB pointer. If the user wishes to set the modifiers for output to SPRINT,SPUNCH, or SERCOM or input through SCARDS, then the second adcon should be zero and the third adcon should point to a four-byte region containing the characters SPRI, SPUN, SCAR, or SERC. The fourth adcon points to a full word containing an MTS line number. If this adcon is zero, then no special line number is assumed.

Example:

```
CALL IOPMOD,(NEWMODS,0,SPRN,0)
•
•
NEWMODS DC A(X'2000') turn on machine carriage
                      control for SPRINT
SPRN    DC C'SPRI'
```

SUBROUTINE DESCRIPTION

Name: LINK,XCTL,LOAD

Purpose: To effect dynamic loading

Calling sequence: OS (I) S type. LINK and XCTL can be called from a FORTRAN program.

Entry: GR1 contains the location of a parameter list:

Word 1: Location of the name of the file or device from which to LINK, XCTL, or LOAD.

Word 2: If non-zero, location of initial ESD (external symbol definition) list which will be appended to the built-in initial ESD list. See the loader writeup for format of table.

Word 3 (LINK or XCTL): Address of a parameter list to be passed in GR1 to the program LINKed to or XCTLed to.

Word 3 (LOAD only): Address of a word of switches

bit 29=1 force private storage

30=1 use system level storage

31=0 use "highest level" storage

=1 use "current level" storage

Other bits must be zero.

Word 4 (LOAD only): If non-zero, is location starting at which the loader is to <u>put</u> the ESD after loading.

Return (LOAD only): GR15 has the location of entry point of what was loaded.

GR0 contains the storage index number (SI#) used:

0 to 7F private
80 to FF system
100 to 17F public

Function of LINK and XCTL

Either entry point will load the program specified using the same "map" device as specified in the initial '$RUN' command. The program loaded will be called in such a way that if it returns (to GR14) then all storage it occupies and all storage it has requested will be released and control will be passed to the next higher level program. The "next higher level program" is the program which called LINK or the program which LINKed to the program which called XCTL.

If XCTL is called it will release all storage used by the calling program before loading the specified program. It will also set things up to return to the program that LINKed to the calling program if the program it loads returns. The parameter list for XCTL may be located in a section of storage that will be released.

Storage allocation

There are two different calls to the GETSPACE routine that a program running under MTS can make to request a block of storage. The first type associates the storage obtained with the current level of LINK so that it is released at the next return or XCTL. The second type associates the storage with the highest level program so it is not released until execution is terminated.

SUBROUTINE DESCRIPTION

Name: LINPG

Purpose: Solve linear programming problems

Calling sequence:
```
OS (I) S type
CALL LINPG(A,Y,M,N,M2,N2,SW)
or its assembly language equivalent.
```

Description

Stated generally, the linear programming problem seeks an extreme value for a linear function subject to linear constraints. With a linear programming problem, by the nature of the mathematics involved, two related mathematical forms of stating the problem always appear. These two forms are customarily designated as the primal and dual forms of the problem. The problem can be stated mathematically in its most general form as

Primal form

(1) Minimize $A(M,1)Y(1) + \ldots + A(M,N+1)Y(N-1)$

subject to the constraints

(2) $A(I,1)Y(1) + \ldots + A(I,N-1)Y(N-1) = A(I,N)$
$I=1,\ldots,M2$

(3) $A(I,1)Y(1) + \ldots + A(I,N-1)Y(N-1) \leq A(I,N)$
$J=N2+1,\ldots,N-1$

(4) $Y(J) \geq 0$
$J=N2+1,\ldots,N-1$

or

Dual form

(1) Minimize $X(1)A(1,N) + \ldots + X(M-1)A(M-1,N)$

subject to the constraints

(2) $X(1)A(1,J) + \ldots + X(M-1)A(M-1,J) = A(M,J)$
$J=1,\ldots,N2$

(3) $X(1)A(1,J) + \ldots + X(M-1)A(M-1,J) \leq A(M,J)$
$J=N2+1,\ldots,N-1$

(4) $X(I) \geq 0$
$I=M2+1,\ldots,M-1$

where A(I,J) are real numbers. Note (2) above are equations while (3) are inequalities.

This statement of the linear programming problem is designated the most general form because any problem whose objective is obtaining an extreme value for a linear function subject only to linear constraints can be put in this form. It is obvious that this can always be done by simply changing the signs of the linear functions and rearranging the rows and columns as required. The roles of primal and dual can always be interchanged in formulating the problem. The machine code requires that the problem be presented in what is designated as the primal form. In particular, this means that

A) The first M2 constraints must be equations, where M2 is between 0 and M-1.

B) The first N2 variables are free to assume either sign, where N2 is between 0 and N-1, and the remaining (N-1-N2) variables must be positive.

The situation with respect to the solvability of the linear programming problem, as well as the general relationship between the primal and dual problems, can be summarized in a general theorem.

LINEAR PROGRAMMING THEOREM

With any given linear programming problem,

1) there exists a finite value V and solution vectors (Y*) and (X*) such that

$$V = \min(A(M,1)Y^*(1)+\ldots+A(M,N-1)Y^*(N-1))$$
$$= \max(X^*(1)A(1,N)+\ldots+X^*(M-1)A(M-1,N))$$

or

2) the constraints for the primal problem are inconsistent, and the constraints for the dual problem are either inconsistent or the dual extreme function is unbounded,

or

3) the primal external function is unbounded, and the constraints for the dual problem are inconsistent.

The machine code is complete in the sense that is provides the solution for a linear programming problem in the event of any of the three possibilities listed in the linear programming theorem. It should be stressed that this machine code does not make any non-degeneracy assumption or stipulate any conditions on the consistency of the constraints or the

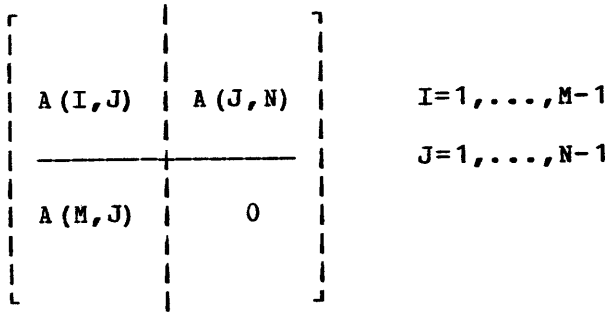
finiteness of the extremal value. The novel algorithm employed in this machine code also has the following properties:

1) It does away with all augmentation for any reason, in particular "artificial variables" for "initial feasible solutions".

2) It does not resort to any perturbation technique or arbitrary cycling of "basis" to cope with the "degeneracy problem", but instead capitalizes on it by concentrating on a greatly restricted subset of rows and columns.

3) It isolates and identifies inconsistent constraints in a simple and natural way.

A general linear programming problem, it should be recognized, is completely specified by the array of numbers, hereafter called the augmented constraint matrix --



and the four parameters: 1) M, the number of rows, which would be the number of constraints and one for the extremal form; 2) M2, the specification that the first M2 constraints are equations; 3) N, the number of columns which would be the number of variables and one for the column of "stipulations" of the constraints; 4) N2, the specification that the first N2 variables can assume arbitrary signs.

Arguments

A First element of the augmented floating point constraint Matrix. This must be a long real packed array of maximum size 150 by 150. The array must be set up column by column, as in FORTRAN.

Y First element of the long real solution vector.

M The integer number of rows in the matrix A. M is also one plus the number of constraints. Must be less than 151.

N The integer number of columns in the matrix A. N is also one plus the number of variables. Must be less than 151.

M2 The first M2 constraints are equations. M2 is between 0 and M-1 inclusive.

N2 The first N2 variables are free to assume either sign. N2 is between 0 and N-1 inclusive. The remaining (N-1-N2) variables must be positive or zero.

SW Short floating point computation switch.

CW SW = 1.0, normal return, Y contains the solution, CT = the value of the extremal function (short floating point).
SW = 2.0, the constraints of the problem are inconsistent, CT= the index of the inconsistent constraint (short floating point).
SW = 3.0, the extremal function is unbounded, CT = the index of the inconsistent constraint of the dual problem (short floating point).

References:

Graves, Glenn W. "A Complete Constructive Algorithm for the General Mixed Linear Programming Problem That Does Not Require Augmentation or Perturbation" doctoral dissertation, University of Michigan, 1962

Graves, Glenn W. "A Constructive Algorithm for the General Mixed Linear Programming Problem" (condensation of doctoral dissertation) Report ATR-64(7040)-1, 30 June 1964, Aerospace Corporation, El Segundo Technical operations, El Segundo, California.

MTS-253/44645-0

12-1-67

SUBROUTINE DESCRIPTION

Name: MOUNT

Purpose: Allow users to mount private volumes.

Calling sequence: OS (I) S type

Entry: GR1 contains the location of a pointer to a parameter list. The parameter list consists of a halfword length of a character string immediately followed by that character string. The character string should be those characters which would immediately follow the "PAR=" if this subroutine were being run as a library file. See the description of the library file *MOUNT, MTS-280/44645, for details.

Exit: RC = 0 volume was mounted successfully
= 4 volume was not mounted.

SUBROUTINE DESCRIPTION

Name: PGNTTRP

Purpose: to set things up so control will be returned to the user on a program interrupt.

Calling sequence: OS R type
Register 13 savearea is not required

Entry: GR0 should contain zero or the location to transfer control to if a program interrupt occurs.

GR1 contains the location of a 72 byte save region in which are stored pertinent information (see below).

Description: A call on this routine sets up a program interrupt exit for one interrupt only. When an interrupt occurs and the exit is taken, it is reset so that another call on this subroutine is necessary to intercept the next program interrupt. A call on this routine with GR0 containing zero resets the exit.

When the program interrupt exit is taken, the first 8 bytes of the save region contains the PSW, and the rest of it contains the general purpose registers in order 0 through 15. The floating point registers remain as they were at the time of the interrupt. GR1 contains the location of the save region.

If, on a call to this routine, the first byte of the region specified by GR1 is X'FF', then in addition to setting up the program interrupt exit, the program is restarted as specified by the PSW in the region, using the GPR's in the region. If the exit routine returns (to the address in GR14) the normal program interrupt processing is invoked.

The SPIE macro is also available and works exactly the same as in OS/360.

SUBROUTINE DESCRIPTION

Printer Plot Routine

The printer plot routine has the following entry points: PLOT1,PLOT2,PLOT3,PLOT4,PLOT14,STPLT1,STPLT2,OMIT, SETLOG.

Purpose: Produce plots in normal output stream.

Calling sequence: Each entry point is called with an OS (I) S type sequence

The standard approach to produce a plot is to call on PLOT1,PLOT2,PLOT3, and PLOT4 in that order.

PLOT1

PLOT1 sets up the information required to construct the graph. The calling sequence is:

CALL PLOT1,(NSCALE(1),NHL,NSBH,NVL,NSBV)

NSCALE(1) A full-word integer vector to supply information about scaling and the number of decimal places to be printed out. If NSCALE(1)=0, the values 0,3,0,3 are used for NSCALE(2) through NSCALE(5) respectively.

NSCALE(2) If this = Y, numbers printed along the y axis are 10**Y times their true value.

NSCALE(3) The number of decimal places printed for Y values.

NSCALE(4) If this = Y, numbers printed along x axis are 10**Y times their true values.

NSCALE(5) The number of decimal place printed for x values.

NHL The full-word integer number of horizontal lines. This number must be 2 or greater. It is one larger than the corresponding NHL in the 7090 plot routine.

NSBH The full-word integer number of spaces between the horizontal lines. This number must be 1 or greater. It is one less than the corresponding NSBH in the 7090 plot routine.

NVL — The full-word integer number of vertical lines. This number must be 2 or greater. It is one larger than the corresponding NVL in the 7090 plot routine.

NSBV — The full-word integer number of spaces between the vertical lines. This number must be 1 or greater. It is one less than the corresponding NSBV in the 7090 plot routine.

Return codes

0 Normal return

4 Improper argument, PLOT1 considered to have been not entered.

PLOT2

PLOT2 prepares the grid and sets up the information required by PLOT3 to place a point correctly in the graph. The calling sequence is

```
CALL PLOT2,(IMAGE,XMAX,XMIN,YMAX,YMIN)
```

If PLOT1 has not been entered by the time PLOT2 is called, the following default configuration is used for PLOT1, to be called the standard grid: NSCALE(1)=0,NHL=6,NSBH=9,NVL=11,NSBV=9 If the SPRINT device is a teletype, NVL is set to 6.

IMAGE — the first byte of an area equal or greater to in length than (NSBH*NHL-NSBH+NHL)*(NSBV*NVL-NSBV+NVL+8)+8 bytes. For the standard grid this is 5567 bytes. This region is used to hold the image of the graph.

XMAX — The largest x value of the points to be plotted.

XMIN — The smallest x value of the points to be plotted.

YMAX — The largest y value of the points to be plotted.

YMIN — The smallest y value of the points to be plotted

The preceding four arguments are either short or long floating point numbers.

Return codes

0 Normal return

8 XMAX≤XMIN or YMAX≤ YMIN. PLOT2 considered not to have been entered.

PLOT3

PLOT3 places the plotting character in the graph for each point (X,Y). The calling sequence:

CALL PLOT3,(BCD,X,Y,NDATA,INT)

BCD	The plotting character to be used.
X	The first element of a floating point vector of X-values.
Y	The first element of a floating point vector of Y-values.
NDATA	The full-word integer number of points to be plotted.
INT	The full-word integer number of bytes between the characteristics of numbers to be used as coordinates. For a short form vector this is 4.
Return codes	0 Normal return
	12 using log scale with negatve numbers (See SETLOG).
	16 PLOT2 not yet entered.

PLOT4

PLOT4 prints the completed graph with values along the X and Y axes and a centered vertical label down the left side. The calling sequence is

CALL PLOT4,(NCHAR,LABEL)

NCHAR	the full-word integer number of characters in the vertical label. If this is 0, no label will be printed.
LABEL	the location or the first character of the label.
	See OMIT for the possibility of deleting grid values and the last graph line.
Return codes	0 Normal return
	20 PLOT2 not yet entered.

PLOT14

PLOT14 allows the user to combine successive calls on PLOT1,PLOT2,PLOT3, and PLOT4 into one call on PLOT14. The calling sequence is:

CALL PLOT14,(NSCALE,NHL,NSBH,NVL,NSBV,
IMAGE,XMAX,XMIN,YMAX,YMIN,BCD,X,Y,
NDATA,INT,NCHAR,LABEL)

See the descriptions of PLOT1,PLOT2,PLOT3, and PLOT4 for the return codes used.

STPLT1

STPLT1 is called by the user who wishes the PLOT routine to inspect his data and then make appropriate calls on PLOT1 and PLOT2. The standard grid size is always used, but the scaling and decimal places to be printed is determined by STPLT1. The user will have to call on PLOT3 and PLOT4 to get a graph printed out. The calling sequence is:

CALL STPLT1,(IMAGE,X,Y,NDATA,INT)

STPLT2

STPLT2 does the work of of STPLT1 and in addition calls on PLOT3 and PLOT4 to obtain a graph. The calling sequence is:

CALL STPLT2,(IMAGE,X,Y,NDATA,INT,BCD,NCHAR,LABEL)

The arguments given to STPLT1 and STPLT2 are described under PLOT2,PLOT3, and PLOT4, as are the return codes used.

SETLOG

SETLOG is called by the user to specify whether he wants a normal, semi-log, or log-log plot. The calling sequence is:

CALL SETLOG,(ARG)

ARG — a byte interpreted as follows:
bit 7 0 y scale is normal
1 y scale is logarithmic
bit 6 0 x scale is normal
1 x scale is logarithmic.
The plotting mode is set initially to normal.

Return code — always 0

OMIT

OMIT is called by the user to specify whether the last graph line, the vertical grid values, or the horizontal grid values will be printed. The calling sequence is:

CALL OMIT,(ARG)

ARG — a full-word integer interpreted as follows: if ARG is positive the function designated by the appropriate bit

is turned off. To turn it back on, make ARG negative and call on OMIT again.

bit 31 horizontal grid values
bit 30 vertical grid values
bit 29 the last graph line

Return code always 0

If one wished to produce a long graph, this could be done by producing the graph in pieces, deleting the horizontal grid values and the last graph line (ARG = 5) for each piece except the last, starting the next graph segment where the last graph line would have been printed. When the last segment was to be printed, OMIT could be called with ARG = - 5 to restore the functions. Initially, all three functions are turned on.

Logical Devices Needed: SPRINT

When using a printer as the SPRINT device, one normally issues a skip to the next page before calling on PLOT4 to start the plot at the top of a page.

SUBROUTINE DESCRIPTION

Name: READ

Purpose: Obtain input record from specified logical unit

Calling sequence: OS (I) S type

Entry: GR1 contains location of a parameter list:

Word 1: A(REGION)

2: A(COUNT)

3: A(MODIFIERS)

4: A(LINE NUMBER)

5: A(UNIT NUMBER)

Exit: RC = 0 Normal return
= 4 EOF

The parameters are described in section 252.

Caller sets up Word 3 parameter and Word 5 parameter, and if indexed is specified, Word 4 parameter.

Subroutine reads record from unit specified into region specified, puts count into place specified, and if sequential, puts line number into place specified.

Notes

1. This subroutine may be called from FORTRAN as well as assembly code. E.g., CALL READ(REG,LEN,0,LUNIT,&30) Note that LEN must INTEGER*2

2. There is a macro of the same name in the system macro library for generating the calling sequence to this subroutine. See writeup in section 255.

Default specifications for units: None

SUBROUTINE DESCRIPTION

Name: REWIND

Purpose: "Rewind" a logical I/O unit in FORTRAN.

Calling sequence: (in a FORTRAN program) CALL REWIND(ARG) where ARG is a full word integer expression between 0 and 9.

Description: If the logical I/O unit specified by ARG is attached to a tape it is rewound. If it is attached to a file it is reset so that the next reference to it will read or write the line specified by the beginning line number given in the $RUN command. In all other cases the call is ignored and a return code of 4 is given.

SUBROUTINE DESCRIPTION

Name: REWIND#

Purpose: Reset a magnetic tape or a file to read from the beginning.

Calling sequence: OS (I) R type

Entry: either (a) GR0 contains a FDUB location (as GETFD returns) and GR1 contains zero.

or (b) GR0 and GR1 contain an 8 character logical I/O unit name, left justified with trailing blanks. [See the $RUN command description for a list of logical I/O unit names.]

Exit: RC=0: Everything ok.

RC=4: Unable to rewind the device specified by GR0 and gr1.

Description: If GR0 and GR1 specify a tape it is rewound. If they specify a file it is reset so that if the next reference to this FDUB or logical I/O unit is sequential, it will read or write the line specified by the beginning line number given when the file was attached. For all other cases a return code of 4 is given.

Note: If the logical I/O unit or FDUB specified by GR0 and GR1 is part of an explicit concatenation, this subroutine affects only the currently active member of the concatenation.

SUBROUTINE DESCRIPTION

Name: SCARDS

Purpose: Read a "card" (input record)

Calling sequence: OS (I) S type

Entry: GR1 contains location of a parameter list:

Word 1: A(REGION)

2: A(COUNT)

3: A(MODIFIERS)

4: A(LINE NUMBER)

Exit: RC = 0 Normal return

= 4 EOF

The parameters are described in section 252.

Caller sets up Word 3 parameter, and if indexed is specified, Word 4 parameter.

Subroutine reads record into region specified, puts count into place specified, and if sequential, puts line number into place specified.

Notes:

1. This subroutine may be called from FORTRAN as well as in assembly language. E.g., CALL SCARDS(REG,LEN,0,LNR,&30) Note that LEN must be INTEGER*2.

2. There is a macro of the same name in the system macro library for generating the calling sequence to this subroutine. See the writeup in section 255.

Default specification for unit Source.

SUBROUTINE DESCRIPTION

Name: SDUMP

Purpose: Produce a dump of any or all of the following:

(1) general registers
(2) floating point registers
(3) a specified region of core storage.

Calling sequence: OS (I) S type

Entry: GR1 contains the location of a fullword aligned parameter list:

Word 1: A(SWITCHES)

2: A(OUTSUBR)

3: A(WKAREA)

[4: A(FIRSTLOC)]

[5: A(LASTLOC)]

RC=0 normal return
RC=4 bad parameters

The parameter words must be aligned on consecutive fullword boundaries. The caller sets up words 1-3 for all calls on SDUMP; words 4 and 5 are also required if the switches indicate that a region of core storage is to be dumped. All parameters must be addresses within addressable core and the first three must be aligned as indicated below, otherwise return with RC=4 occurs.

Parameter Description

Word 1 is the address of a fullword aligned fullword of switches that govern the content and format of the dump produced. The switches are assigned as follows:

Bit 24: on if a core region is to be dumped

25: on if floating point registers are to be dumped

26: on if general registers are to be dumped

27: on if long output records (130 characters) are to be formed; off if short output records (70 characters) are to be formed

28: on if double spacing is desired; off if single spacing is desired

29: on if EBCDIC conversion of the core region is desired

30: on if MNEMONIC conversion of the core region is desired

31: on if HEXADECIMAL conversion of the core region is desired

The default case (all switches off) produces a dump as though bits 24,25,26, and 31 were on. Furthermore, if bit 30 (mnemonics) is on bit 31 (hexadecimal) is implied. Note that bits 24,25, and 26 specify what is to be dumped, bits 27 and 28 specify the page format, and bits 29,30, and 31 specify the interpretation(s) to be placed on the region of core specified. Bits 29 through 31 have significance only if bit 24 is on.

Word 2 must give the half-word aligned address of a subroutine (e.g., SPRINT) that causes the printing, punching, etc. of ouput line images formed by SDUMP

Word 3 is the address of a doubleword aligned area of 400 bytes that may be used by SDUMP as a work area.

Word 4 is the address of the first byte of a core region to be dumped. There are no boundary requirements for this address.

Word 5 is the address of the last byte of a core region to be dumped. There are no boundary requirements for this address; however, an address in Word 5 which is less than that in Word 4 will cause a return with RC=4.

Output Formats

Registers:

General and floating point registers, if requested, are always given in labelled hexadecimal format. The length of the output record is governed by the setting of bit 27 of the switch.

Core Storage:

Although any combination of switches is acceptable, the appearance of the dump output for a region of core is determined as follows:

1. If, and only if, the mnemonic switch is on the unit of core storage presented in each print item is a halfword aligned halfword.

2. If, and only if, the mnemonic switch is off and the hexadecimal switch is on (through intent or default), the unit of core storage presented in each print item is a fullword aligned fullword.

3. If, and only if, the mnemonic and hexadecimal switches are off but the EBCDIC switch is on, the unit of core storage presented in each print item is a doubleword aligned doubleword.

In all cases the output includes (1) the entire core storage unit (halfword, fullword, or doubleword) in which the first specified core location (parameter word 4) is found, (2) the entire core storage unit in which the last location (parameter word 5) is found, and (3) all intervening storage. That is, the first and last printed items of a core storage dump may include up to a maximum of seven core bytes more that actually requested in the parameter list.

If mnemonics are requested and SDUMP discovers a byte that cannot be interpreted as an operation code, then, instead of a legal mnemonic, the characters "XXXX" appear directly below the hexadecimal presentation of the halfword in core that should have contained an operation code. When this occurs, the mnemonic scanner jumps ahead as though the illegal operation code specified an RR type instruction (i.e., two bytes) and tries to interpret the byte at the new location as an operation code, and so forth. Any mnemonic print line that contains the "XXXX" for at least one of its entries is also marked with a single "X" directly below the line address that prefixes the hexadecimal presentation of that same region of core. (The mnemonic conversion routine includes the universal instruction set and also those instructions exclusively used by Model 67). To facilitate the location of particular items in the output, line addresses always have a zero in the least significant hexadecimal position and column headers are provided. The column headers give the value of the least significant hexadecimal digit of the address of the first byte in each print item.

A line of dots is printed to indicate that a region of core storage contains identical items. The core storage unit used for comparisons is halfword, fullword, or doubleword, depending on the type(s) of conversion specified. In all cases the core storage unit corresponding to last item printed before the line of dots and that for the first item after the line and all intervening core storage units have identical contents. The last line is always printed (even if all of its entries exactly match the previously printed line).

SUBROUTINE DESCRIPTION

Name: SERCOM

Purpose: Print an error comment (output record)

Calling sequence: OS (I) S type

Entry: GR1 contains location of a parameter list:

Word 1: A(REGION)

2: A(COUNT)

3: A(MODIFIERS)

[4: A(LINE NUMBER)]

Exit: RC = 0 Normal return

= 4 Output device full

The parameters are described in section 252.

Caller sets up all parameters. Word 4 of parameters is needed only if the modifiers specified indexed or return line number (peel). A word 3 of zero is assumed equivalent to a Word 3 containing the location of a zero word.

Subroutine writes out a record of length specified from region specified.

Notes:

1. This subroutine may be called from FORTRAN as well as assembly code. E.g., CALL SERCOM(REG,LEN,0) Note that LEN must be INTEGER*2

2. There is a macro of the same name in the system macro library for generating the calling sequence to this subroutine. See the writeup in section 255.

Default specification for unit: Master sink (terminal)

SUBROUTINE DESCRIPTION

Name: SETIOERR

Purpose: Allow the user to regain control when I/O errors that would otherwise be fatal occur during execution.

Calling sequence: OS (I) S type.
Register 13 save area is not required.

Entry: GR1 contains the location of an adcon containing either the location of a subroutine to transfer to when an I/O error occurs or zero, in which case the error exit is reset.

When the error routine is called, registers 0 and 1 both contain what was in GR13 upon entry to the I/O routine, i.e., the location of the save area in which the I/O routine saved registers at the time of the call.

If the error routine returns, return is made from the I/O routine as if no error had occurred.

The error exit is reset at a normal termination of execution (return or call on SYSTEM), or just prior to executing the next $RUN command, if abnormal termination of execution occurred. (call on ERROR, attention or program interrupt, etc).

SUBROUTINE DESCRIPTION

Name: SETPFX

Purpose: To set the prefix character issued during execution as the first character of every input or output line.

Calling sequence: OS (I) S type
callable from Fortran and assembly code.

Entry: GR1 contains the location of a parameter list
word 1 is the location of the prefix characters
word 2 is the location of a fullword count of the number of characters.

Exit: GR0 contains the previous prefix character, right-justified with leading hexadecimal zeros.

Restriction: Currently only one character may be used. Hence only the first character at the location specified is used.

Example:
```
OLD = SETPFX('/',1)
```

SUBROUTINE DESCRIPTION

Name: SPRINT

Purpose: Print a "line" (output record)

Calling sequence: OS (I) S type

Entry: GR1 contains location of a parameter list:

Word 1: A(REGION)

2: A(COUNT)

3: A(MODIFIERS)

[4: A(LINE NUMBER)]

Exit: RC = 0 Normal return

= 4 Output device full

The parameters are described in section 252.

Caller sets up all parameters. Word 4 of parameters is needed only if the modifiers specified indexed or return line number (peel). A word 3 of zero is assumed equivalent to a Word 3 containing the location of a zero word.

Subroutine writes out a record of length specified from region specified.

Notes:

1. This subroutine may be called from FORTRAN as well as assembly code. E.g., CALL SPRINT(REG,LEN,0) Note that LEN must be INTEGER*2

2. There is a macro of the same name in the system macro library for generating the calling sequence to this subroutine. See the writeup in section 255.

Default specification for unit: Sink

Name: SPUNCH

Purpose: Punch a "card" (output record)

Calling sequence: OS (I) S type

Entry: GR1 contains location of a parameter list:

Word 1: A(REGION)

2: A(COUNT)

3: A(MODIFIERS)

[4: A(LINE NUMBER)]

Exit: RC = 0 Normal return

= 4 Output device full

The parameters are described in section 255.

Caller sets up all parameters. Word 4 of parameters is needed only if the modifiers specified indexed or return line number (peel). A word 3 of zero is assumed equivalent to a Word 3 containing the location of a zero word.

Subroutine writes out a record of length specified from region specified.

Notes:

1. This subroutine may be called from FORTRAN as well as assembly code. E.g., CALL SPUNCH(REG,LEN,0) Note that LEN must be INTEGER*2.

2. There is a macro of the same name in the system macro library for generating the calling sequence to this subroutine. See the writeup in section 255.

Default specification for unit: None (terminal usage)
PUNCH (batch usage)

SUBROUTINE DESCRIPTION

Name: SYSTEM

Purpose: Stop the world - I want to get off

Calling sequence: None - just branch
(callable from both assembly code and FORTRAN)

Effect: Returns control to MTS to terminate execution. Execution terminated in this manner cannot be RESTARTed. The comment "EXECUTUION TERMINATED" is printed for a normal termination of execution like this.

MTS-253/66315-0

12-1-67

SUBROUTINE DECRIPTION

Name: WRITE

Purpose: Write output record on specified logical unit

Calling sequence: OS (I) S type

Entry: GR1 contains location of a parameter list:

Word 1: A(REGION)

2: A(COUNT)

3: A(MODIFIERS)

4: A(LINE NUMBER)

5: A(UNIT NUMBER)

Exit: RC = 0 Normal return

= 4 Output device full

The parameters are described in section 252.

Caller sets up all parameters, except that Word 4 parameter must be set up only the modifiers specified indexed.

Subroutine writes out a record of length specified on unit specified from region specified.

Notes:

1. This subroutine may be called from FORTRAN as well as assembly code. E.g., CALL WRITE(REG,LEN,0,LNR,3). Note that LEN must be INTEGER*2.

2. There is a macro of the same name in the system macro library for generating the calling sequence to this subroutine. See the writeup in section 255.

Default specification for units: None

MACRO LIBRARIES

This section describes the macros available in the system macro library (*SYSMAC) and how to construct a private macro library.

System Macro Library - *SYSMAC

The system macro library (*SYSMAC) contains all of the macros described in this section (MTS-255). It may be utilized by specifying it for logical I/O unit 0 when running the assembler:

```
$RUN *ASMBLR; 0=*SYSMAC  SCARDS=.......
```

Some of the macros available are essentially the same as in IBM's Operating System (OS/360): they have the same effect (except where specified) as their OS equivalents, but generate code appropriate for interfacing with MTS, not OS. They are listed below. For descriptions, see IBM's publication "Supervisor and Data Management Macro - Instructions", form C28-6647. Descriptions of other macros are on the following pages.

OS - equivalent Macros

ABEND	(no parameters, same as ERROR - q.v.)
CALL	
DCB	
DCBD	
FREEMAIN	no SP no MF
GETMAIN	no SP no MF
LINK	only EP or EPLOC
LOAD	only EP or EPLOC
RETURN	
SAVE	
TIME	(no parameters - returns HH:MM.SS in GR0-GR1, MM-DD-YY in GR2-GR3)
WTO	
WTOR	no ecb address
XCTL	only EP or EPLOC

Note: If *SYSMAC is used, the following additional macro names cannot be defined by the user -- IHBERMAC, IHBINNRA, IHBOPLST, IHBRDWRS, IHBRDWRK, IHB01, IHB02, #IOHPRLS, #IOHOCCK, #FIX, #IOHERCK, #LNXC, #MTSIOCP, #MTSIOST, #MTSIOG , #MTSIOEF, #MTSIOPS

MACRO DESCRIPTION

Name: ACCEPT

Purpose: To "accept" an image so that the next Read Format or Look At Format using IOH/360 will see a new image. This does not apply to FORTRAN format I/O.

Prototype: [LABEL] ACCEPT [UNIT]

where UNIT is the name of a full-word aligned full word containing either a logical unit number or a FDUB pointer which specifies the unit whose image is to be accepted. If omitted, SCARDS is assumed.

Note: A call (by IOH/360) to RCLOSE causes the "accept" switch to be set for the appropriate logical I/O unit; a call to LCLOSE just sets the return code to zero and returns to IOH/360. A user may decide to do some Look At Formats and then decide that he wants to read the <u>next</u> image— to do this he should ACCEPT the current image and the Look At or Read the next image with LKFMT or RDFMT.

MACRO DESCRIPTION

Name: BAS,BASR

Purpose: To assemble the BRANCH-AND-STORE and BRANCH-AND-STORE-REGISTER instructions which are opcodes 4D (hex) and 0D (hex) respectively on the IBM 360 Model 67.

Prototype:

```
[LABEL]   BASR    R1,R2
[LABEL]   BAS     R1,D2(X2,B2)
```

Usage: These instructions are essentially the same as the BAL and BALR instructions except that the high-order byte of the branch register is zeroed out (i.e., the instruction length code (ILC) and condition code (CC) are not inserted) in normal PSW mode or extended PSW mode with either no translation on or 24 bit translation on [MTS normally runs with 24 bit translation on]. See the Model 67 Functional characteristics manual (A27-2719) for details.

MACRO DESCRIPTION

Name: DFAD,DFSB,DFMP

Purpose: To simulate double-precision floating point using the double precision hardware instructions. These macros will add, subtract, or multiply two contiguous long floating point registers by a double precision (16 byte) operand and place the result in the appropriate floating point registers.

Prototype: [LABEL] DFxx REG,ADR,TEMP

Usage: The contents of ADR (8 bytes long aligned on a double-word boundary) and ADR+8 (8 bytes long aligned on a double-word boundary) are added to, subtracted from, or multiplied by the contents of long floating registers REG and REG+2. Note that REG cannot be 6 as there is no floating register 8. TEMP specifies a scratch area which will be used: in the case of DFAD and DFSB. The scratch area must be 16 bytes long, aligned on a double-word boundary; in the case of DFMP the scratch area must be 64 bytes long, aligned on a double-word boundary.

A double precision operand may be considered as two long floating operands. Both operands have a characteristic and a mantissa; the characteristic of the second long operand is 14 less than the characteristic of the first long operand.

These macros use the hardware operations ADD, ADDR, SDD, SDDR, MDD, and MDDR. [These instructions are RPQ number M24391]

MACRO DESCRIPTION

Name: DFIX, EFIX

Purpose: To convert a floating point number (in a floating point register) to an integer (in a general register).

Prototype: [LABEL] {DFIX / EFIX} FPR,GR[,WA=workarea]

where
- FPR is the floating-point register
- GR is the general register.
- WA designates a work area of two double words aligned on a double word boundary. If omitted, the macro will allocate an in-line work area.

Description: DFIX converts a long precision floating point number (8 bytes); EFIX converts a short precision floating point number (first four bytes of floating point register). The contents of the specified floating point register are restored at the end of the macro call. Note that it is possible to convert a floating point number that is too big to fit (as an integer) in a general register - e.g., the floating point number 1.2E30 is obviously too big. No attempt is made to signal this as an error.

MARCO DESCRIPTION

Name:	DISMOUNT
Purpose:	Set up calls to dismounting routines
Prototype:	[LABEL] DISMOUNT 'MESSAGE'
Parameters:	'MESSAGE' is the pseudo-device-name of the volume to be dismounted
Example:	DMNT DISMOUNT '*TAPE*'
Description:	A call will be set up to DISMOUNT. (See MTS 280/24627). The macro generates literal constants, hence literal adressability must be preserved.

MACRO DESCRIPTION

Name: ENTER

Purpose: To generate prolog code for the entrance to a subroutine.

Prototype: [LABEL] ENTER REG[,SA=savearea][,LENGTH=len][,TREG=tempreg]

where
- REG is the register to be established as a base register. Should not be 15.
- SA specifies the location of a save area to use. If SA is omitted, a call to GETSPACE for a save area of length specified by LENGTH is made.
- LENGTH specifies the length of the save area to be obtained if SA is omitted. If LENGTH is not specified, 72 is used.
- TREG specifies a temporary register which is used. If omitted, register 15 is used. TREG should not be the same as REG.

Examples

```
SUBR   ENTER  12
F      ENTER   9,SA=SAVARR
G      ENTER  15,TREG=12
```

Effect: Causes code for the following to be produced:

1. Generates USING *,REG
2. Establishes REG as base register
3. If SA is omitted calls GETSPACE to get save area LENGTH long
4. Establishes forward and backward links

MACRO DESCRIPTION

Name: EXIT

Purpose: Re-establish calling program's save area and return with a return code in register 15.

Prototype: [LABEL] EXIT RC[,MF=FS]

where RC is a self-defining term which is the return code. MF=FS specifies that the save area pointed to by register 13 is to be released.

Examples

```
      EXIT   4
OUT   EXIT   0
```

Usage: This requires that the save area was properly linked on entry to the subroutine, such as is done by the ENTER macro. If the ENTER macro was used and it obtained space via GETSPACE, this space can be released by specifying MF=FS.

MACRO DESCRIPTION

Name: FLOAT

Purpose: To convert the contents of a general register or a full-word aligned area in storage into a floating-point number and leave the converted number in a floating point register.

Prototype: [LABEL] FLOAT ARG1,ARG2

where

ARG1 can either be a general purpose register or a full-word aligned location; if ARG1 designates a general register then the argument must be enclosed in parentheses.

ARG2 is the floating point register into which the results are placed.

Note: Contents of register zero are destroyed if ARG1 specifies a storage location.

Literal addressability is required.

Examples:

```
FLOAT  A,2      Float the contents of A and put in FPR 2
FLOAT  (6),6    Float the contents of GR6 and put in FPR 6
```

MACRO DESCRIPTION

Name: GETSPACE

Purpose: To assemble a call to the GETSPACE subroutine

Prototype: [LABEL] GETSPACE [LENGTH,] T=type

where LENGTH is a self-defining term specifying the number of bytes of storage wanted. If omitted, the length is assumed to be in general register 1.

T specifies the type of space wanted. type should be 0,1,2, or 3. See description of GR0 contents in GETSPACE writeup [MTS-253/27635].

Example: GETSPACE 8192,T=3

Macro Calls to IOH/360

A set of Macro Definitions has been written to allow the user to painlessly call IOH/360 since the calling sequence is somewhat ghastly and one may not like forming the calling sequence manually more than a few times.

The following Macros have been defined:

RDFMT - 'read format'

LKFMT - 'look at format'

WRFMT - 'write format'

PCFMT - 'punch format'

PRFMT - 'print format'

MOREIO- 'continue parameter list'

ONEIO - 'continue parameter list one element at a time'

ENDIO - 'terminate I/O-processing'

IOP - 'I/O-parameter' (acts like ONEIO or MOREIO depending on the number of arguments given)

REFMTC- 'repeat call'

SERFMT- 'print formatted error comment'

Whenever the term LIST is used in the following descriptions, it will have the following meaning:

> The LIST is the parameter list --- i.e. the arguments on which IOH/360 is to work as directed by the FORMAT. The LIST may be either a single element or a list of elements each to be taken as an argument or part of a block-argument. If there is more than one element in the LIST, then the whole LIST must be enclosed in parentheses.

More specifically:

```
          <LIST> ::= <atomic argument> | (<listp>)
         <listp> ::= <list element> | <listp>,<list element>
  <list element> ::= <atomic argument> | <block sublist>
 <block sublist> ::= <atomic argument>,...,<atomic argument>
<atomic argument> ::= <legal Assembler Language expression>
```

INITIAL CALL TO IOH/360

The initial call to IOH/360 is made using the Macros RDFMT,LKFMT,WRFMT,PCFMT, or PRFMT. The general form of these five Macro Calls is as follows (the 'read format' call will be used as an example):

```
RDFMT FORMAT,LIST,OPEN=MYOPEN,CLOSE=MYCLOSE,EOF=MYEOF,LUNIT=MYUNIT,
      ERROR=MYERROR,SECT=MYSECT,POOLSW= 0 or 1,NC=ARGNC,TYPE=MYTYPE
```

FORMAT refers to the user-supplied format which will tell IOH/360 how to interpret fields going to or coming from the line image. LIST has been described previously --- LIST may be empty (i.e. it may be omitted). The other parameters in the Macro Call (i.e. the keyword arguments) are optional. Any or all may be deleted --- if any of the keyword arguments appear, they may appear in any order. If the user wishes to use one of the keyword arguments, he must place some symbol (which is either system-defined or user-defined) in place of the "MY"-symbols as shown above.

```
example: PRFMT FMT1,(A,B,A+100,...,A+200,0),POOLSW=1,TYPE=S,
                CLOSE=MFCLOSE,OPEN=MLOPEN,SECT=PQSECT,NC=12
```

THE KEYWORD OPTIONAL ARGUMENTS

OPEN and CLOSE

OPEN — Indicates the name of a user-supplied routine which IOH/360 will call when it needs a new line image; the routine so named may be internal to the calling program or external to it; in the latter case it must have been defined in an EXTRN statement.

CLOSE — Indicates the name of a user-supplied routine which IOH/360 will call when it needs to put out a line image (i.e. release a line

image); the routine so named may be internal to the calling program or external to it; in the latter case the name must have been previously defined in an EXTRN statement.

NOTE: If the user is going to supply one of the pair of an OPEN or a CLOSE, then he should supply both (especially in the case of the 'read format' usage). See the IOH/360 writeup about the details of writing an OPEN or CLOSE routine.

ERROR returns and END-OF-FILE returns

The user may specify that on each return from IOH/360, a check of the return code is to be made. If either the EOF or ERROR arguments appear, then, a check of the return code is to be made --- and the generation of the appropriate code will be made to route the return to the point specified by the user. NOTE: IOH/360 will return to the user with a non-zero return code only if the correct flag bits have been previously set by a call to SETIOHER (see IOH/360 writeup for details).

EOF indicates the name of a user-defined or system-defined routine to handle END-OF-FILE returns from IOH/360; the routine may be internal or external to the calling program --- if external, it must have been declared external in an EXTRN statement. NOTE: IOH/360 will allow an EOF return only if the user has set the appropriate flag by calling on SETIOHER --- if this switch has not been set, return will be made to the system routine SYSTEM with the comment: ALL INPUT DATA HAS BEEN PROCESSED AT LOCATION XXXXXX.

ERROR indicates the name of a user-defined or system-defined routine to handle ERROR returns from IOH/360; the routine may be internal or external --- if external, it must have been been declared so in an EXTRN statement. NOTE: IOH/360 will allow an ERROR return only if the user has set the appropriate flag by calling on SETIOHER --- if this flag is not set, return will be made to the system routine ERROR, with the comment: ERROR RETURN TO SYSTEM.

SECT: Placement of Parameter Lists

SECT indicates the name of a Control Section in which the parameter lists generated by the current Macro Call are to be placed. The SECT argument need not be given on every call. The effect of a previous SECT argument is retained until the appearance of a new SECT argument. Initially, Macro Calls are generated with in-line parameter lists with a branch around them. The user may cause these parameter lists to be placed in the Control Section designated by the last SECT argument. If the last or present SECT argument given is an asterisk, "*", then in-line parameter lists will be coded with a branch around them.

POOLSW: Accumulation of Sublist-Count Addresses

POOLSW each time a Macro Call is expanded, half-word constants may be generated which form the counts for the lists and sublists. The user may cause such half-words to be accumulated so that half-words previously generated are not generated and the previously defined half-words are used (i.e. the macro expander keeps track of names it has assigned to certain half-word constants that it has generated previously). The proper use of this switch allows one to avoid generating redundant half-word constants --- it is a space-saver in this respect. The POOLSW argument acts as a binary switch. It may take on the values 0 or 1. Each time it is explicitly used (whether it is given an assignment 0 or 1) in a Macro Call, all previous half-word constant reference accumulation is lost; i.e. the half-word constant pool begins to be built from scratch. The initial value of POOLSW is 0. If the assignment to POOLSW is 1, then new constant pools are generated for each Macro Call thereafter until the setting of POOLSW is changed. If the assignment to POOLSW is 0, then an accumulation of half-word constants starts at the present Macro Call and continues until the assignment of POOLSW is explicitly given again (i.e. explicitly a 0 or 1).

NC: Use of Previous Lists

NC the user may assign "names" to the Macro Calls which he uses by using the NC parameter. He then may refer to such an assigned name in a later Macro Call to use the same format and list defined in the Macro Call which originally defined the name (this

allows the user to use the same format and parameter list for many Macro Calls; it saves space). The NC argument can be any string of characters up to eight characters in length. Only twenty-five such (different) names may be used in a single assembly. The following happens on encountering an unomitted NC argument. A search is made of the list which contains previously defined Macro Calls using the NC argument. If the present NC argument has not been used before, then an entry for the present Macro Call is made in the table and processing proceeds as if the NC argument was omitted. If the present NC argument is already in the list, then both the FORMAT and the LIST of the present Macro Call are ignored. The calling sequence for the present Macro is so arranged as to point to the parameter list which contains the FORMAT and the LIST of the Macro Call which originally defined the present NC argument. When the NC argument is used in an REFMTC Macro Call, it specifies that in addition to using the LIST and FORMAT of a previous Macro Call, the OPEN and CLOSE arguments are also to be used (see REFMTC description following).

TYPE: The Type of Adcons Generated for the List

TYPE — The lists that the Macro Processor generates may be in one of two forms. The first type consists of full-word adcons (i.e. type-A adcons). The second type of list consists of half-word adcons (i.e. S-type adcons). Initially a type-A list is generated. One may set the value of the keyword "TYPE" to either A or S to generate one or the other type of list. The type so selected will also be used in all succeeding calls until the TYPE is changed.

LUNIT: The Logical Unit Number

LUNIT — The user may specify a logical unit number (between 0 and 9) to which or from which images are to be passed. If LUNIT is omitted as a keyword argument, then the routines in the table below will be used for input and output of images. If the LUNIT keyowrd is present, the logical unit so named (i.e. LUNIT=0 ; "0" is the logical unit in this case) will be used to get input images (using the READ subroutine) or release output images (using the WRITE subroutine). The LUNIT argument may also be the address of a FDUB pointer. The image length in such cases is 256.

DEFAULTS FOR 'OPEN' AND 'CLOSE'

If the OPEN or CLOSE parameters are omitted and the LUNIT keyword is omitted, the Macros will generate adcons with the names of the appropriate SYSTEM-supplied OPEN and CLOSE routines. They are as follows:

MACRO CALL	OPEN	CLOSE	ROUTINE CALLED AND LENGTH OF IMAGE
RDFMT	ROPEN	RCLOSE	SCARDS-80 columns (not called if previous input image has not been 'accepted')
LKFMT	LOPEN	LCLOSE	SCARDS-80 columns (not called if previous input image has not been 'accepted')
PRFMT	POPEN	PCLOSE	SPRINT- 132 columns
PCFMT	PCOPEN	PCCLOSE	SPUNCH- 80 columns
WRFMT	POPEN	PCLOSE	SPRINT- 132 columns
SERFMT	SEROPEN	SERCLOSE	SERCOM- 132 columns

NOTE: PRFMT should be used for transmitting an image to some kind of printer. WRFMT should be used for transmitting images to any I/O-device capable of receiving the image. PRFMT and WRFMT actually generate the same calls.

NOTE: a call on RCLOSE or ACCEPT causes the input image transmitted by a previous call to LOPEN or ROPEN to be 'accepted'.

ADDITIONAL FEATURES

Whenever IOH/360 finds a zero as an argument address, it terminates I/O processing. The appearance of the zero element may appear anywhere in the argument list where reference to an I/O argument may be placed. If IOH/360 works its way through an argument list and does not find the zero element, it returns to the user. The user has the option of returning or not returning to IOH/360 with another argument list. If he does not return or makes a new initial call to IOH/360 (the initial types of call have been defined above), then he "loses" any remaining image left in the image buffer. If he returns with another parameter list, I/O processing will continue from the place where it was when IOH/360 previously returned to the user. This process can continue as long as the user wants to do so --- IOH/360 just keeps on performing I/O-conversion until the zero element is reached. The return to IOH/360 can be generated with four types of macros. they are:

```
MOREIO
ONEIO
ENDIO
IOP
```

MOREIO-with this Macro, the user can generate a new argument list, can specify EOF, ERROR, SECT, NC, POOLSW, and TYPE arguments, and has the option of telling IOH/360 to use a new format instead of continuing with the original format. The general form of the call is:

```
MOREIO LIST,EOF=MYEOF,ERROR=MYERROR,SECT=MYSECT,CFMT=FORMAT,
        POOLSW=0 or 1,NC=NCARG,TYPE=MYTYPE
```

If CFMT is used all knowledge pertaining to the previous format will be lost --- i.e. IOH/360 uses the new format just as if it were the original one. In particular, knowledge of placement of parentheses and their associated multiplicities in the previous format is lost. With the CFMT feature, the user may build a format as he goes along during execution (he needs to have only a part of the format present at any one time) --- one thing to remember is that the user must supply enough format each time for the number of arguments in the associated parameter list or terminate each such format with an asterisk (i.e. '*'). Note that if the NC argument refers to a previously compiled Macro Call, then the LIST and CFMT arguments of the present call are ignored and the LIST and FORMAT (or CFMT) of the previous Macro Call which used that NC argument will be used.

ONEIO- with this Macro, the user may give IOH/360 one argument at a time

much like the STR (store and trap) on the 7090. He may also specify the EOF and ERROR arguments. The form of the Macro Call is :

```
ONEIO ARG,EOF=MYEOF,ERROR=MYERROR
```

If ARG is a zero (i.e. ONEIO 0), I/O-processing is terminated.
NOTE: If ARG is omitted, register 1 is assumed to contain the argument address.

ENDIO- with this Macro, one can terminate an I/O list. It is equivalent to ONEIO 0.

IOP- to transmit either a single argument or a single blocked-pair, one may use the IOP Macro. The EOF and ERROR keyword arguments may be used in this macro call.

```
IOP ARG is equivalent to ONEIO ARG

IOP ARG1,...,ARG2 is equivalent to MOREIO (ARG1,...,ARG2)
```

REFMTC: How to "Execute" Previous Calls

The REFMTC Macro is used to "execute" previously defined initial-type Macro Calls. The general form of an REFMTC call is:

```
REFMTC NC=NCARG,ERROR=MYERROR,EOF=MYEOF
```

The NC-argument must be present and refers to a previously-defined initial-type Macro Call using that NC-argument. The effect of this type of call is to set up code which will use the OPEN and CLOSE arguments of the initial-type call (even if they were defaulted), in addition to using the FORMAT and LIST of that call. The use of this Macro Call would, for example, allow one to have many places in a program to read a card using the exact same FORMAT, LIST, and OPEN and CLOSE options. It is a space-saver in this respect.

NOTE

So that the user's names do not conflict with the names generated during a Macro Call, all names generated during Macro processing start with a #-sign and have a number immediately following the #-sign. All such names have at least three (3) characters in them. Also, all global set-symbols used during Macro Call compilation begin with a #-sign.

The names of the Macros that are used in expanding the IOH/360 macros described above are as follows:

```
#IOHPRLS
#IOHOCCK
#IOHERCK
```

Thus the user should not define his own macros with any of the above names.

EXAMPLES

```
PRFMT FORMAT1,(A,B,C,...,C+8,A+5,A+10,...,A+15,0),OPEN=IOPEN,
      CLOSE=ICLOSE
```

The above call is equivalent to the following set of calls:

```
PRFMT FORMAT1,OPEN=IOPEN,CLOSE=ICLOSE
IOP A
IOP B
IOP C,...,C+8
IOP A+5
IOP A+10,...,A+15
ENDIO
```

MACRO DESCRIPTION

Name:	MOUNT
Purpose:	Set up calls from execution to mounting routines.
Prototype:	[LABEL] MOUNT 'MESSAGE'
Parameters:	'MESSAGE' is the character string which would immediately follow "PAR=" if the subroutine were called in command mode.
Examples:	MOUNT 'M123 ON 9TP, PNAME=Test, ''TEST TAPE'',RING IN'
Description:	A description of the mount subroutine is given in MTS 280/44645. The macro generates literal constants, hence literal addressability must be preserved. Note also double primes are used to produce one prime in actual text.

MACRO DESCRIPTION

Name: SCARDS,SPRINT,SPUNCH,SERCOM,READ,WRITE

Purpose: To assemble calls to the system I/O routines of the same names.

Prototypes: Type 1: The user specifies a region and length.

```
[LABEL] {SCARDS} REG [,REGL][,MODIFS][,LNR][,EXIT=exitseq]
        {SPRINT}
        {SPUNCH}
        {SERCOM}

[LABEL] {READ } UNIT,REG[,REGL][,MODIFS][,LNR][,EXIT=exitseq]
        {WRITE}
```

Type 2: The user specifies a comment.

```
[LABEL] {SPRINT} 'comment'[,MODIFS][,LNR][,EXIT=exitseq]
        {SPUNCH}
        {SERCOM}

[LABEL]  WRITE  UNIT,'comment'[,MODIFS][,LNR][,EXIT=exitseq]
```

where REG is the location of the region to be read into or written from. It may be expressed as a symbol on the number of a register (which contains the location of the region) in parentheses.

REGL specifies the length. It may be either the name of a half-word containing the length or a self-defining term which is the length, or the number of a register (which contains the length) in parentheses. If omitted, L'REG is assumed.

MODIFS stands for several parameters separated by commas. Each consists of the name of an MTS modifier (see section MTS-210) preceded by either an at-sign ("@") or an at-sign not-sign ("@¬").

LNR is the location of a full-word aligned full word containing the line number.

EXIT specifies exits to be taken for non-zero return codes. If exitseq is a single symbol, then any non-zero return code will cause a branch to this symbol. If exitseq is a parenthesized list of symbols, then a return code of 4 will cause a branch to the first symbol, a return code of 8 will cause a branch to the second symbol, and so on. If a return code larger than that corresponding to the last symbol occurs, then a branch to the last

symbol will take place.

UNIT — specifies the corresponding parameter to be given to READ or WRITE, either the numbers 0 through 9 or the location of a full-word aligned full word containing a FDUB pointer.

Examples:

```
          SPRINT    '1PROGRAM TO DIDDLE AROUND'
   LOOP   SCARDS    REG,LEN,EXIT=EOF
          READ      5,INREG,L,EXIT=(EOF,OUCH)
          SPRINT    LINE,80
          SCARDS    REGION,LENG,@I,@PEEL,EXIT=DONE
          SPUNCH    (1),80
```

Other forms: If MF=L is attached to any of the calls listed above, then only the parameter list will be generated. If the parameters are replaced by MF=(E,LISTADR)[,EXIT=exitseq] then LISTADR is assumed to be the location of a parameter list, and only the code to call the subroutine will be generated.

MACRO DESCRIPTION

Name: SLT

Purpose: To assemble a SLT (search list) instruction - opcode A2

Prototype:

[LABEL] SLT M1,L2,D2B2

where M1 is the mask to be used
L2 is the count to be used (actual count - not IBM count)
D2B2 should be represented in a form suitable for assembly into an S-type adcon, i.e., D(B), where B is the register containing the location of the list region and D specifies the displacement from the front of the block of the word containing the chain address.

Example: SLT 10,2,LINKO(0)

Usage: A description of the SLT instruction is available at the Computing Center. [The SLT instruction is RPQ number F13800]

MACRO DESCRIPTION

Name: SWPR

Purpose: To assemble an SWPR (swap register) instruction - opcode A3

Prototype:

```
[LABEL]   SWPR   GR,FR
```

where GR is the even general register of an even-odd pair
FR is a floating-point register.

Examples:

```
    SWPR   6,4
L   SWPR   0,0
```

Usage: The SWPR instruction exchanges the contents of a floating point register and a pair of general registers. [The SWPR instruction is RPQ number M24390]

MTS Assembly Language Testing Macros

A set of macros has been written which allows the user to write assembly programs without intermediate printing, and to add or delete such printing as is useful during the debugging phase without disturbing the code already written.

The following macros have been defined:

```
#AT          -to define testing point(s)
#DISPLAY     -to print an arbitrary length string of bytes
#COMMENT     -to print a comment
#GENREGS     -to print the general registers
#FLTREGS     -to print the floating point registers
#GOTO        -to transfer control within the testing macros
#STOP        -to halt execution
#ENDTEST     -a necessary evil
```

Explanations and restrictions:

#AT

```
Prototype:          [pplabel]  #AT  proglabel
                    [pplabel]  #AT  (proglabel1,proglabel2,...)
```

The first occurrence of an #AT macro call inserts into the calling program a 24-26 byte link to a setup routine. Therefore the first #AT must follow relative address formation and be executed before any testing point is reached. 'pplabel' will only be defined in the first #AT call. Then it will be the first instruction in the link sequence. 'proglabel' must label an instruction in the user's program. There is no default for 'proglabel'.

```
Examples:                   #AT  T1
                PLABEL      #AT (TESTING,PRLABEL,HERE)
```

#DISPLAY

```
Prototype:        [ttlabel}   #DISPLAY stringname[,abcon[,type]]
```

This macro prints 'abcon' bytes beginning at 'stringname' in either ebcdic ('type' equals BCD, EBCD, or EBCDIC) or hexadecimal ('type' is any other string) conversion. There is no default for 'stringname'. Default for 'abcon' is L'('stringname'). Default for 'type' is hexadecimal.

```
Examples:       TLABEL      #DISPLAY  ARRAY
                            #DISPLAY  ARRAY,,BCD
                            #DISPLAY  SAVEAREA,72
```

COMMENT

Prototype: [ttlabel] $COMMENT 'characterstring'

Forty bytes is the maximum length of 'characterstring'. 'characterstring' must contain at least one character.

```
Examples:       TLABEL1     #COMMENT  'THIS COMMENT PRINTED'
                            #COMMENT  'SO DOES THIS.'
```

#GENREGS

Prototype: [ttlabel] #GENREGS

```
Examples:       TLABEL2     #GENREGS
                            #GENREGS
```

#FLTREGS

Prototype: [ttlabel] #FLTREGS

```
Examples:       TLABEL3     #FLTREGS
                            #FLTREGS
```

#GOTO

Prototype: [ttlabel] #GOTO ttlabel

This macro transfers control within the testing macros. 'ttlabel' may be a label on any testing macro except #AT and #ENDTEST.

```
Examples:       TLABEL4     #GOTO   TLABEL3
                            #GOTO   TLABEL2
```

#STOP

Prototype: [ttlabel] #STOP

This macro stops execution with an operation interrupt. Everything except GR15 is available for $ALTERing. $RESTARTing resumes execution. This macro has limited use in batch jobs.

```
Examples:       TLABEL5     #STOP
                            #STOP
```

#ENDTEST

Prototype: #ENDTEST

One call on this macro must be included in the user's program, and it must physically follow all other testing macro calls.

CAUTIONS:

1) The testing macros produce many names of the form @TESTxxx. These should be avoided by the user.

2) This facility operates through program interrupts and will not behave if the user is processing his own interrupts.

3) An ERROR RETURN with GR14 containing a location in @TESTITP indicated that the facility found an interrupt which did not belong to it. In the dump only GR0 is useful and it points to an 18 word save area which contains the conditions of the program at the time of the interrupt. Words 1 and 2 contain the PSW and words 3 to 18 contain GR0 to GR15.

4) The facility will alter the program condition code immediately before the instruction named by an #AT macro. There is no other effect upon the execution of the program.

Example of usage:

```
          USING *,15
          STM   0,15,SAVEREGS


          PRINT NOGEN
          #AT T1
          #AT T2
          #COMMENT 'THESE LOCATIONS HAVE NOT BEEN SET'
          #DISPLAY ARRAY,,BCD
          #DISPLAY ARRAY
          #GENREGS
          #FLTREGS
MORE      #COMMENT 'ONE LAST COMMENT'
          #AT T3
          #GOTO MORE
          #AT (T4,T5)
          #DISPLAY ARRAY,50
          #AT T6
          #STOP
          #COMMENT 'RESTARTED...'
          #ENDTEST


T1        BCR   1,0
T2        BCR   2,0
T3        BCR   3,0
T4        BCR   4,0
          MVI   ARRAY,X'48'
          MVI   ARRAY,C'X'
          MVC   ARRAY+1(49),ARRAY
T5        BCR   5,0
T6        BCR   6,0
          LM    0,15,SAVEREGS
          BR    14
SAVEREGS  DS    16F
          DS    CL3
ARRAY     DS    CL100
          END
```

MTS-255/63625-0

12-1-67

Output from example:

```
# AT T1
#
# AT T2
# THESE LOCATIONS HAVE NOT BEEN SET
# ARRAY,100
# A**&*******& **&* *p*****************************************&*&******
# *********************************
# ARRAY,100
#       C102
#       C1 00035018 0000CB2C 00035070 000350A4 4000D7AE 00000000 0001898
# 00029E98 81038018 0000CE38 0000CE5C 00008768 0100CB2C 0000CB2C 000350D
# 5000CCBE 00000000 00000004 0000CB2C 0002AD20 0000CE38 0000CE5C 0000876
# 0100CB2C 00000005
# GENERAL REGISTERS
#       00029FAC     00029F48     00000000     40017986
#       00000000     00029F40     00000000     00029FAC
#       00000000     00029038     000140D6     00012030
#       00011030     00029F54     00014CAC     00035018
# FLOATING POINT REGISTERS
# 0000000000000000  0000000000000000  0000000000000000  0000000000000000
# ONE LAST COMMENT
#
# AT T3
# ONE LAST COMMENT
#
# AT T4
#ARRAY,50
#       C1 00035018 00000B2C 00035070 000350A4 4000D7AE 00000000 0001898
# 00029E98 81038018 0000CE38 0000CE5C 00008768 0100CB2C
#
# AT T5
# ARRAY,50
#       E7 E7E7E7E7 E7E7E7E7 E7E7E7E7 E7E7E7E7 E7E7E7E7 E7E7E7E7 E7E7E7E
# E7E7E7E7 E7E7E7E7 E7E7E7E7 E7E7E7E7 E7E7E7E7 E700CB2C
#
# AT T6
# STOP...TYPE '$RESTART' TO CONTINUE
#PROGRAM INTERRUPT. PSW = FF450001 40041276
#$RESTART
  RESTARTED...
#EXECUTION TERMINATED
```

Structure of a Macro Library

A macro library consists of a directory and the macros.

A. The directory:

1. Each entry of the directory contains the name of a macro in columns 1-8 and the line-number of the macro definition header of the corresponding macro in columns 10-16. (Both the name and the line-number must be left justified with trailing blanks).

2. The line-number of the first entry in the directory must be 1.

3. The terminating entry in the directory is a string of eight zeroes in columns 1-8.

B. The macros:

1. The line-number of the macro-definition header of each macro must be a positive integral number.

2. The first macro follows the last entry in the directory.

EXAMPLE:

```
$NUMBER 1,1
BASR     10
BAS      20
00000000
$NUMBER 10,.1
         MACRO
&LABEL   BASR  &REG1,&REG2
&LABEL   BALR  &REG1,&REG2
         MEND
$NUMBER 20,.1
         MACRO
&LABEL   BAS   &REG1,&LOC
&LABEL   BAL   &REG1,&LOC
         MEND
```

MTS-270-1

12-1-67

PHONE NUMBERS - DATA SET DIRECTORY

FREQUENTLY CALLED NUMBERS

Computing Center Model 67 Ports:

TELETYPE	(through 2702)	763-0300	(hunt-on-busy)
	(through data concentrator)	764-0200	(hunt-on-busy)
2741	(through 2702)	763-0510 763-0530	[same as 1050]
	(through data concentrator)	764-0206	
1050	(through 2702)	763-0510 763-0530	[same as 2741]

COMPLETE DIRECTORY

[ITEMS IN BRACKETS ARE NOT YET CONNECTED]

ORDERED BY NUMBER-

NUMBER	DATA SET	ROOM	BLDG	WHAT CONNECTED TO	ANSWER-BACK
482-0794	103A		WR	33ASR(PORTABLE)-BLDG.2218	UMISTRSWRAYP
482-7915	103A	130	WR	35ASR-BLDG.2041	UMCPDETWRAYP
483-2652	103A		WR	33ASR-BLDG.2213	UMISTIRWRAYP
721-8110	103A	2254	FLU	35ASR-FLUIDS ENG. LAB.	UM NUENG AAA
761-1947	103A		CRLT	1050	
3-0139	103A	UNIV	TWRS	33ASR-LOG. OF COMP. GRP.	UMLOGICCGAAA
3-0149	103A		EE	33ASR-BLL RAD. LAB.	UMEERALAB AAA
3-0164	103A	2113	SP	35ASR-SPACE PHYSICS	UMSPCPHYSAAA
3-0167	103A	1101	NUB	33ASR	UMGEOSBSLAAA
3-0170	103A	HWY	SAF	33ASR	UMHYSRI A AA
3-0190	201A	2518	EE	LINC-8	
3-0193	103A	19	P.S.	33ASR	UM H P C AAA
3-0196	103A	HOSP	ADMN	33ASR(PORTABLE)	UMBUHOSADAAA
3-0300 TO 3-0315	103A		CC	TELETYPE PORTS OF 2702 (HUNT-ON-BUSY LINES)	
3-0316	103A		CC	35ASR-MACHINE ROOM	UM CMPC A AA
3-0510	103A		CC	2741/1050 PORT OF 2702	
3-0530	103A		CC	2741/1050 PORT OF 2702	
3-0563	103A	1008	CHEM	33ASR	UM CHEM A AA
3-0570	201A		CC	DECAFACE - SDR1	
3-0571	201A		CC	DECAFACE - SDR2	
3-0572	201A		CC	DECAFACE - SDR3	
3-0573	201A		CC	DECAFACE - SDR4	
3-1175	103A	2518	EE	LINC-8	
3-1176	103A	2518	EE	LINC-8	

NUMBER	DATA SET	ROOM	BLDG	WHAT CONNECTED TO	ANSWER-BACK
3-2189	201A		MHRI	PDP-8	
3-2196	103A	106	DENT	33ASR	UM DENTL AAA
4-0200 TO 4-0207	103A		CC	DECAFACE PORTS (HUNT-ON-BUSY LINES)	
4-1538	103A	UNI	COMP	35ASR-ARCH. RES	UMARCHRESAAA
4-1571	103A	2514	EE	35ASR	UMSYSENGEEA
4-2273	103A	2116	IST	33ASR	
4-2329	103A		MH	1050 - LANGUAGE LAB	
4-2345	103A		CC	2701 - TEL2	
4-3249	103A		CRLT	33ASR	MIT DUM1 DUM1
4-3288	103A	1031	CC	1050	
4-4149	103A	B101	SPH	1050	
4-4169	103A	3216	EE	35ASR	UMCMENGAAA
4-4208	202C		CC	DECAFACE	
4-4259	103A		CC	2741-MACHINE ROOM	
4-4279	103A	108	CC	[2741-BASEMENT]	
4-4289	103A	231	COOL	33ASR	UMCOLYEEMAAA
4-4297	202C		CC	DECAFACE	
4-4398	103A		CC	2701 - TER1	
4-4439	103A	230F	WE	35ASR	UM WENG C AAQ
4-4483	X403		CC	DECAFACE	
4-4494	X403A		CC	DECAFACE	

NUMBER	DATA SET	ROOM	BLDG	WHAT CONNECTED TO	ANSWER-BACK
4-4496	103A		CC	DECAFACE - MAIN FRAME	
4-6155	103A	1528	UES	33ASR	UMUNELSCLAAA
4-6199	103A	2518	EE	33ASR	UMSYSEE B AA
4-6569	103A	1525	MHRI	35ASR	UM MHRI A AA
4-6591	X403A		MH	(LANGUAGE LAB)	
4-7144	201A	UNIV	TWRS	DEC PDP-7-LOG. OF COMP.GRP	
4-7145	201A	230F	WE	DEC 338	
4-7146	103A	230F	WE	DEC 338-MAINFRAME	
4-7147	103A	230F	WE	DEC 338-33ASR	
4-7539	103A	229	WE	33ASR(PORTABLE)	UMESUMCONFAA
4-8270	103A	2001	MHRI	35ASR	UM MHRI B AA
4-9525	103A	108	CC	35ASR-BASEMENT	UM CMPC B AA
4-9526	103A		CC	33ASR-MACHINE ROOM (PORTABLE)	UM CMPC C AA
4-9527	103A		CC	35ASR - MACHINE ROOM	UM CMPC D AA
4-9528	103A	230F	WE	35ASR	UM WENG A AA
4-9529	103A	228H	WE	33ASR-PORTABLE(HERZOG)	UM WENG B AA
4-9564	103A		CC	1050 - MACHINE ROOM	
4-9570	103A	2601	SPH	35ASR	UM SPH A AA

LIBRARY FILE DESCRIPTIONS

This section contains descriptions of many of the library files available to MTS users.

LIBRARY FILE DESCRIPTION

Name: *ASA

Contents: The object module of a program to convert lines with ASA printer control characters in column 1 to lines with machine carriage control in column 1 (which can be directly printed).

Usage: This file should be referenced by a $RUN command.

Logical I/O units referenced:

SCARDS - Source of the lines with ASA carriage control

SPRINT - printer (*SINK* in batch) or place to put the output lines.

Examples:
```
$RUN *ASA; SCARDS=-PRINT
     (SPRINT defaults to *SINK*)
$RUN *ASA; SCARDS=-PRINT SPRINT=LISTING
```

Note: If the output of *ASA is put in a file, then when the file is finally listed, the machine carriage control modifier should be specified:

```
$COPY LISTING TO *SINK*@MCC
```

LIBRARY FILE DESCRIPTION

Name: *ASMBLR

Contents: The initial object module of the Operating System F-level Assembler.

Purpose: Assemble System/360 assembly language source programs.

Usage: The F assembler is invoked by an appropriate RUN command specifying *ASMBLR as the file where object cards are found.

Logical I/O units referenced:

SCARDS	The source program which is to be assembled.
SPRINT	The assembly listing and symbol reference table listing.
SPUNCH	The resulting object module.
0	A library of macro definitions. This should be set to *SYSMAC unless the user wishes to use his own macro library.
1	The file to contain the object module if it is to be executed. This is needed only if the parameter LOAD is set.
2	A second library of macro definitions, described below. This is needed only if the parameter CON is specified.

Examples: $RUN *ASMBLR
(SCARDS,SPRINT,SPUNCH default to *SOURCE*, *SINK*, *PUNCH*, resp.)

```
$RUN   *ASMBLR;  SPRINT=*SINK*  SCARDS=*SOURCE*
       0=*SYSMAC  1=-LOAD  PAR=NODECK,LOAD,LINECNT=99

$RUN   *ASMBLR;  SPUNCH=LD  SCARDS=CDS 0=*SYSMAC
```

Description: This assembler is IBM's Operating System F-level Assembler modified to run under MTS, and therefore offers the full facilities of that language. For a complete description of the language see the manual "IBM System/360 Operating System Assembler Language", form C28-6515-4. Two additions have been made to the language. The COM pseudo operation may

have a symbol in the label field. The common section generated is given this name exactly as in FORTRAN named common. Also the operand field of a COPY pseudo operation can be any FDname as described in MTS-210. If parameters are to be given, the PAR= sequence must be last in the sequence of specifications, as shown in the example above. The parameters are:

DECK — The object module is placed on the device specified by SPUNCH.

LOAD — The object module is placed on logical unit 1.

LIST — A listing of the program is put on the device specified by SPRINT.

XREF — A cross-reference table of symbols is put on the device specified by SPRINT.

RENT — The assembler checks for a possible coding violation of program reenterability.

CON — A macro library with the same structure as *SYSMAC is placed as the file specified as logical unit 2. This library will be searched before the library on 0 is searched, permitting the over-riding of system macro definitions or the concatenation of macro libraries.

LINECNT=nn — This parameter specifies the number of lines to be printed between headings in the listing. The permissible range is 01 to 99 lines.

TEST — TESTRAN symbol table information is produced. (See *SDS writeup for usage.)

The prefix NO is used to turn off parameters LOAD, LIST, XREF, RENT, CON, and TEST. If no options are specified, the assembler assumes the following default entries:

```
PAR=NOLOAD,DECK,LIST,XREF,NORENT,
    NOCON,NOTEST,LINECNT=56
```

Note: The assembler usage writeup will be found in section MTS-510.

LIBRARY FILE DESCRIPTION

Name: *ASMEDIT

Contents: The object module of a program (1) to process card images suitable for input to the MTS Assembler into standard format; that is, name field starting in column 1, operation field starting in column 10, operand field starting in column 16, and following comments delimited by a blank. Continuation characters in column 72 are not disturbed. (2) to sequence the card images produced with a four character program identification code followed by four numeric characters.

Purpose: To edit MTS Assembler input source decks into easily read form. This feature is especially desirable to clean up decks containing modifications made from terminals.

Usage: The editing program is invoked by a RUN command specifying *ASMEDIT as the file containing the object module.

Logical I/O units referenced:

SCARDS	A file containing card images suitable for input to the MTS Assembler.
SPRINT:	A listing of the edited deck is given on the specified device.
SPUNCH:	The edited deck is output to the specified file or device.

Example:

```
$RUN *ASMEDIT; SCARDS=PGM SPRINT=-LIST SPUNCH=FGME
$RUN *ASMEDIT; SCARDS=PGM
   (SPRINT defaults to *SINK*,SPUNCH to *PUNCH* (in batch))
```

Description: *ASMEDIT edits card images suitable for input to the MTS Assembler into standard format for the F-level assembler. It is useful for producing a neatly aligned deck from decks containing modifications made from terminals. The first card processed by *ASMEDIT must contain a four character deck identification code in columns 73-76 and a starting sequence number in columns 77-80; viz.

```
PUNT0000
|      |
73     80
```

*ASMEDIT will continue numbering, incrementing by one, from the starting sequence number.

LIBRARY FILE DESCRIPTION

Name: *ASMERR

Contents: The object module of a program to scan an assembly listing and find flagged lines and their associated error comments.

Purpose: To print the errors and diagnostics noted in a F-level assembly listing.

Usage: The error listing program is invoked by a RUN command specifying *ASMERR as the file containing the object module.

Logical I/O units referenced:

SCARDS	A file containing an assembly listing
SPRINT	Error comments are given on the specified device

Example:
```
$RUN *ASMERR;SCARDS=-PTR
    (SPRINT defaults to *SINK*)
```

Description: *ASMERR lists on SPRINT the erroneous statements and diagnostics noted in an F-level assembly listing. If no errors are detected an appropriate comment is printed. SCARDS must refer to a file, not a device.

LIBRARY FILE DESCRIPTION

Name: *BATCH

Contents: The object module to monitor remote batch entry.

Usage: The program is invoked by a $RUN command specifying *BATCH as the object file.

Logical I/O units referenced:

SCARDS - file or device containing records to be entered as an MTS job.

Examples:

```
$RUN *BATCH
     (SCARDS defaults to *SOURCE*)
$RUN *BATCH; SCARDS=AFILE
```

Description: The content of SCARDS's reference is treated as any "batch" job run in MTS. A "receipt number" by which the user may pick up the output is returned to the user. The job's output may be picked up at the computing center when ready (at the time of this writing, usually the morning following its entry) and should be retrieved within one week.

The first statement entered into SCARDS must be $SIGNON

If SCARDS references a device (such as the user's terminal), rather than a file, the following are applicable:

1. Only 480 characters of information will be accepted (six 80 byte lines, twelve 40 byte lines, etc.)
2. If a line of zero length is entered, the line pointer is decremented by one line, that is, the previous line is deleted.

LIBRARY FILE DESCRIPTION

Name: *BCDEBCD

Contents: The object module of the BCD to EBCD conversion program.

Purpose: Convert cards punched on an 026 card punch (BCD) to the equivalent card codes as if punched on an 029 card punch (EBCD).

Usage: The BCD to EBCD conversion program is invoked by an appropriate RUN command specifying *BCDEBCD as the file where the object cards are found.

Logical I/O units referenced:

SCARDS	BCD lines to be converted.
SPRINT	Listing of converted lines.
SPUNCH	EBCD lines resulting from the conversion.

Examples:

```
$RUN *BCDEBCD; SPUNCH=MYFILE
    (SCARDS defaults to *SOURCE*,SPRINT to *SINK*)
```

Description: The following conversion is applied to all input lines:

BCD CARD CODE	CHARACTER	EBCD CARD CODE
0-4-8	(%	12-5-8
12-4-8	) ¤	11-5-8
4-8	' @	5-8
12	+ &	12-6-8
3-8	= #	6-8
all others		unchanged

This conversion maps those characters which had a dual symbolism (scientific and commercial) on the 026 keypunch to the appropriate 029 keypunch scientific code. Note that on an 026 keypunch, there is no way to represent the following characters:

% ¤ @ & #

as this program will convert them to their scientific equivalents.

LIBRARY FILE DESCRIPTION

Name: *CATALOG

Contents: Object module of program to print a list of files for the current user.

Usage: This file should be referenced by a $RUN command with *CATALOG as the object file.

Logical I/O units referenced:

SPRINT	List of files is put on specified file or device.

Example: $RUN *CATALOG

Description: *CATALOG lists on SPRINT the name of every file belonging to the current user and a summary of the amount of file space he is using.

LIBRARY FILE DESCRIPTION

Name:	*COINFLIP
Contents:	The object module of the Coinflip program.
Purpose:	Demonstration.
Usage:	The program is invoked by the conversational command: $RUN *COINFLIP
Description:	The program is self-describing.

LIBRARY FILE DESCRIPTION

Name: *CONVSNOBOL

Contents: A SNOBOL4 program that reads and executes SNOBOL4 statements.

Purpose: To allow conversational use of SNOBOL4.

Usage: This file contains commands to run the SNOBOL4 program. It should be specified as the source (see $SOURCE command description).

Logical I/O units referenced:

(Specified in $RUN command in the file)

5: Input for SNOBOL4 initial compilation
6: Output for SNOBOL4 error comments
8: Input of conversational SNOBOL4 statements
9: Output of conversational SNOBOL4 statements

Example: $SOURCE *CONVSNOBOL

Description: The SNOBOL program in this file reads (from *MSOURCE*) SNOBOL statements. If a statement is preceded by a number, the compiled form of the statement is stored in the element of the array PROG with that subscript. If the statement is not preceded by a number i is executed immediately and the code is not saved unless the statement was labeled. To begin execution of stored statements the user should execute an unconditional transfer to one of them. If the SNOBOL4 compiler detects an error in the statement entered the comment "COMPILATION ERROR" is typed. When the user terminates execution (by entering an end of file) the error comments for the last such statement are typed. If the SNOBOL4 interpreter detects an error, execution is immediately terminated with an error comment and a string dump if one was requested.

Example session: (Input is underlined)

```
$SOURCE *CONVSNOBOL
$RUN *SNOBOL4;5=*CONVSNOBOL(3) 6=*DUMMY*(1,20)+*MSINK*
  8=*MSOURCE* 9=*MSINK*
EXECUTION BEGINS
SUCCESSFUL COMPILATION
ENTER SNOBOL STATEMENTS
 OUTPUT = "HELLO"
HELLO
```

```
£      (end of file)
NORMAL EXIT FROM SNOBOL AT LEVEL 0
                    •
                    •
                    •
```

LIBRARY FILE DESCRIPTION

Name: *DISKDUMP

Contents: The object module of the DASD dump program.

Purpose: Print the contents of specified portions of a 2311, 2314, or 2301 in a hexadecimal dump format.

Usage: The DASD dump program is invoked by an appropriate RUN command specifying *DISKDUMP as the file where the object cards are found.

Logical I/O units referenced:

SCARDS	Control lines specifying the disks and areas on the disks to be dumped.
SPRINT	The dump listing.

Examples: $RUN *DISKDUMP; SCARDS=*SOURCE* SPRINT=PTR2

Description: The DASD dump program will print in hexadecimal dump format the specified region of any DASD . All records on the DASD within the specified region are printed, with the exception of the R0 records. Each record is read with a READ COUNT, KEY AND DATA command, and the entire record is printed out - beginning with the count field. The format of each control line is:

Positions	Contents
1-4	Physical DASD on which volume is mounted e.g., D291 e.g., D291
5-8	Starting cylinder address in hexadecimal, e.g., 00A0
9-12	Starting track address in hexadecimal, e.g., 0000
13-16	Ending cylinder address in hexadecimal, e.g., 00A5
17-20	Ending track address in hexadecimal, e.g., 0009

As many control lines as desired can be entered. Execution terminates when an end-of-file condition is returned by SCARDS.

All other processing on a DASD being dumped is locked out while that DASD is being printed, so that the contents being dumped cannot be changed in mid-dump, so to speak.

LIBRARY FILE DESCRIPTION

Name: *DISMOUNT

Contents: The object module of the volume dismounting program.

Purpose: Allow users to dismount private volumes.

Usage: The program is invoked by a run command specifying *DISMOUNT as the file where the object cards are to be found.

Prototype: $RUN *DISMOUNT; PAR=*pseudodevicename*

where: *pseudodevicename* is the pseudo-device name specified when the volume was mounted.

Examples: $RUN *DISMOUNT;PAR=*TAPE*

Logical I/O Units Referenced: none.

Description: The volume will be removed and (hopefully) placed in its appropriate spot in the storage rack. See *MOUNT description for further details. If there are any outstanding uses of the pseudo-device name (such as SCARDS) the volume will not be actually dismounted until all of them are released.

The following rules describe the interaction between *MOUNT and *DISMOUNT.

1. If *MOUNT is run (either explicitly or by calling MOUNT during execution) with a pseudo-device name specified which is already active, then the device type must be the same as the previous type and all outstanding references to the pseudo-device name will be changed to refer to the new volume.
2. If *MOUNT is run with a pseudo-device name which has been previously used but for which *DISMOUNT has been run, then the new device type may be anything legal and any outstanding references to the old pseudo-device name are unchanged. In general a use of *DISMOUNT signifies the end of a particular use of a pseudo-device name and a use of *MOUNT for an active pseudo-device name signifies a new volume for a previous use.

The following example may clarify this. Suppose the following commands were issued during an MTS session:

```
$RUN  *MOUNT;  PAR=G000  ON 7TP PNAME=*TP* MY TAPE
```

```
$RUN -OBJ; SCARDS=*TP*
```

If during execution of the program in -OBJ, *MOUNT is called (presumably by using the macro or subroutine MOUNT) and the pseudo-device name *TP* is specified then the device type must be 7TP and a new reel will be mounted on the same tape drive, which SCARDS will still refer to. However if during execution of the program in -OBJ, *DISMOUNT is called for *TP* and then *MOUNT is called for it, the device type may be any legal type, SCARDS will still refer to the old tape, and any new use of the name *TP* (for example as a parameter to GETFD) will refer to the new volume. In this case the original tape reel will not actually be removed until execution is terminated.

LIBRARY FILE DESCRIPTION

Name: *DOUBLE

Purpose: To generate the correct (rounded) double-precision floating point representation of a given decimal number. If 'x' is the decimal representation, then the approximate range of possible values of 'x' is

$$.388E-61 < |x| < 7.21E+75$$

[Note: Double-precision is not long precision]

Usage: Input to this program is on SCARDS; output may be printed or punched and printed. Printed output is on SPRINT; punched output is on SPUNCH. To specify that one wishes punched output, the parameter saying such should be given---i.e. PAR=PUNCH. Any error comments (such as underflow or overflow) will be produced on SERCOM---no error is fatal. The user inputs number (starting in column 1) in any type of floating point external format. Examples:

```
10.1
10.1E+0
10.1D+0
10.10000
101E-1
101-1
10.1-1
```

All numbers (or special characters such as . , D E and + and -) until either a blank, illegal floating point format, or illegal character occurs will be taken as part of the input number. Output will be in the form of two hexadecimal numbers each sixteen digits in length. If the user specifies that punched output is to be given, then the first 15 columns after the first blank or first illegal character will be used as a comments field in the punched output. Punched output is or the form which can be input to the assembler---i.e.

```
DC  X'16 digits',X'16 digits' comments
```

Examples: user input is underlined

```
#$RUN *DOUBLE
#EXECUTION BEGINS
 1.0
```

4110000000000000 3300000000000000
3.14159265358979323846264338327950288419716790202 PI
413243F6A8885A30 338D313198A2E037

LIBRARY FILE DESCRIPTION

Name: *DRAW

Contents: The object module of line drawing program

Purpose: Allow user to create line drawings using the 2250 display unit. Up to 31 separate displays may be saved in a file and later retrieved.

Usage: This program is invoked by an appropriate RUN command specifying *DRAW as the file where the object cards are found.

Logical I/O units referenced:

SCARDS	file from which displays are to be read
SPUNCH	file in which displays are saved

Examples:
```
$RUN *DRAW; SCARDS=DRAWING  SPUNCH=DRAWING
$RUN *DRAW; SCARDS=DRAWING  SPUNCH=PCH1
```

Description: The user at the 2250 display console interacts with the *DRAW program by means of the light pen and programmed function keyboard (PFK). The following keys on the PFK have been given the indicated meaning:

key 0 -> enter point-plotting mode

key 1 -> enter basic character mode

key 2 -> enter large character mode

key 3 -> enter position-cursor mode

key 4 -> enter vector-plotting mode

key 10 -> start scan to find light pen

key 22 -> enter erase mode

key 30 -> read new display (via "SCARDS")

key 31 -> save the current display (via "SPUNCH")

Pressing any other key will cause a display of these instructions to appear. In addition two "alternate code" keys have been given these meanings:

"cancel" -> blank screen in preparation for a new drawing

"end" -> terminate execution of the *DRAW program.

To plot a point on the screen the user must enter point-plotting mode by pressing key 0. (Once a mode is entered, the program remains in that mode until the mode is changed by pressing another key). Pressing key 10 initiates a display of X's. By pointing the light pen at the spot on the screen where the point is to be plotted and pressing the light pen foot switch to activate the light pen, the user "plots" the point. Successive points may be plotted by alternately pressing key 10 and activating the light pen. To draw lines, vector plotting mode must be entered. Key 10 and the light pen must then be used as described above. The position of the light pen denotes the terminal point of the line. The initial point of the line is the most recent point (or terminal point) drawn. Characters (either basic or large size) may be entered in the drawing by entering the appropriate character mode, pressing key 10 and activating the light pen to indicate where in the drawing the characters are to be inserted. This causes a cursor to appear on the screen and up to 10 characters may be entered from the console. If more than 10 characters are desired, the above action can be repeated. To delete lines or points from a drawing, erase mode is entered by pressing key 22 and the object to be deleted is identified with the light pen. To delete characters, enter either erase or position-cursor mode and identify the character with the light pen. This will cause a cursor to appear beneath the character. The character may then be erased by pressing the space bar on the console.

Displays may be retrieved or saved by pressing keys 30 and 31.

LIBRARY FILE DESCRIPTION

Name: *EBCDBCD

Contents: The object module of the EBCDIC-to-BCD translation program.

Purpose: Translate lines from EBCDIC to 7090 Augmented BCD.

Usage: This program is normally invoked with a $RUN command specifying *EBCDBCD as the object FDname. It can also be loaded and called as a subroutine (without parameters).

Logical I/O units referenced:

SCARDS	For EBCDIC lines to be translated.
SPUNCH	Augmented BCD lines resulting from translation.
SPRINT	Listing of the lines resulting from translation.

Example:
```
$RUN *EBCDBCD; SCARDS=OLDFILE SPUNCH=NEWFILE
     (SPRINT defaults to *SINK*)
```

Description: Lines of any length are read sequentially (not indexed) from SCARDS, translated, and written to SPUNCH. Each output line is also prefixed by a blank character (for carriage control) and written to SPRINT. Output lines are written sequentially unless the "indexed" modifier is used. When an end-of-file is encountered from SCARDS, the program returns.

The following translation is applied to each character of each input line:

EBCDIC input		BCD output
Character	Card code	
Digits		Unchanged
Upper-case letters		Unchanged
Lower-case letters		Upper-case equivalents
Control characters		Blank
. , $ * - / ; | ¬ _ "		Unchanged
+	12-6-8	12
&	12	0-7-8
(	12-5-8	0-4-8
)	11-5-8	12-4-8
'	5-8	4-8
=	6-8	3-8
<	12-4-8	12-6-8
>	0-6-8	6-8
%	0-4-8	2-8
$	0-7-8	12-0
:	2-8	5-8
#	3-8	0-2-8
@	4-8	11-0
All others		Blank

LIBRARY FILE DESCRIPTION

Name: *FILEDUMP

Contents: The object module of the file dumping program.

Purpose: To dump the contents of a file or device in hexadecimal.

Usage: The file-dumping program is invoked by the appropriate RUN command specifying *FILEDUMP as the file where the object cards for the file dumper are found.

Logical I/O Units Referenced:

SCARDS - the file or device to be dumped

SPRINT - the file or device where the output is to be placed.

Parameters: The output on SPRINT is usually 64 columns per line; the user may vary the length of the output line by the use of the LENGTH parameter. The LENGTH may vary between 1 and 132 inclusive.

Examples:

```
$RUN *FILEDUMP; SCARDS=MYOBJ
     (SPRINT defaults to *SINK*)

$RUN *FILEDUMP; SCARDS=-OBJ SPRINT=-PRINT
PAR=LENGTH=128
```

Sample Run:

```
$RUN *FILEDUMP;SCARDS=-FILE(1,2)
#EXECUTION BEGINS

 **LINE-FILE DUMPER
 LINE NO.       1    ;  80 CHARACTERS
 02C5E2C4404040404040400030404000001C4E4D4D7C64040400000000000400007DC
 E2C5D9C3D6D4404002000000404040404E2C3C1D9C4E2404002000000404040404
 4040404040404040C4D4D7C6F0F0F0F1
 LINE NO.       2    ;  80 CHARACTERS
 02C5E2C4404040404040400030404000004C7C5E3C6C4404040020000004040404
 D9C5C1C4404040400200000040404040E2D7D9C9D5E3404002000000404040404
 4040404040404040C4D4D7C6F0F0F0F2
#EXECUTION TERMINATED
```

LIBRARY FILE DESCRIPTION

*FILESCAN

Contents: The object module of the file scan program.

Purpose: Locates a line in a file according to a given format and prints the line and line number.

Usage: The filescan program is invoked by an appropriate RUN command specifying *FILESCAN as the file where the object cards are found.

Logical I/O units referenced:

SCARDS	Format and file name input.
SPRINT	Requests and line and line number of selected line.
SERCOM	Error comments.

Description: *FILESCAN will first print "ENTER FORMAT" on SPRINT. There are three types of format terms: skip, transfer, and character string. These are entered via SCARDS. The skip term consists of the letter S followed by an optional minus sign followed by a string of decimal digits. The value of the digit string is the number of columns to be skipped. The transfer term consists of the letter T followed by a string of decimal digits. The value of the digit string is the column in the lines of the file to be scanned where the next skip or character string term will start. The character string term is delimited by primes. The characters in the string are compared with the appropriate columns of the lines of the file to be scanned. A maximum of 20 character strings may be used in one format. If a prime is desired in the character string, it may be represented by two consecutive primes. (Note: Unless it is the last character in a character string, each prime reduces the maximum number of character strings by one.) Commas may be used to separate format terms, but they are not necessary. The format is terminated by the first blank not in a character string or by the end of the line. If the first character entered is a >, it is assumed that the characters immediately following name a file and that the previous format is to be used. If "CONTINUE" (or "C") is entered, the scan is continued with the next line in the same file using the same format.

After the format is entered, *FILESCAN will print "ENTER FILE NAME" on SPRINT. The file name is entered via SCARDS and may be followed by a line number range and/or modifiers as for GETFD. The first blank or end of line terminates the file name.

An end-of-file when a file name is requested will cause a format request. An end of file when a format is requested will cause termination.

Example: If it was desired to find lines containing "LA" in columns 10 and 11 and "R4," in columns 16-18 of the files FILE and FILE1, the input (underlined) and requests might be:

```
#$RUN *FILESCAN
#EXECUTION BEGINS
  ENTER FORMAT
 T10'LA',S4'R4,'
  ENTER FILE NAME
 FILE
      114                LA     R4,INPUT
  ENTER FORMAT
 CONTINUE
      123                LA     R4,1(,R4)
  ENTER FORMAT
 >FILE1
       12                LA     R4,=A(WHERE)
  ENTER FORMAT
 ENDFILE
#EXECUTION TERMINATED
```

LIBRARY FILE DESCRIPTION.

Name: *FORTRAN

Note: *NEWFORT rather than *FORTRAN currently contains the FORTRAN G compiler. See the description of *NEWFORT, filed alphabetically in this section.

LIBRARY FILE DESCRIPTION

Name: *FORTEDIT

Contents: The object module of a program to convert FORTRAN statements in free-format form to fixed-format form.

Purpose: Allow user to write FORTRAN statements in free-format form. The output consists of "fixed-format cards" so that they can be processed by the current FORTRAN compiler.

Usage: The program is invoked by the appropriate RUN command specifying *FORTEDIT as the file where the object cards are found.

Logical I/O units referenced:

SCARDS	free-format input lines to be converted
SPUNCH	fixed-format output lines for processing by FORTRAN IV compiler.

Examples: $RUN *FORTEDIT;SCARDS=FTNPRGM SPUNCH=-PROG

Description:

Initial lines

If a line is the initial line of a statement, it may have a statement number. The first character of a statement number should occur within the first five columns and will terminate at column 6 or at nonnumeric characters (blanks not included), which ever occurs first. The text of a statement begins just at the first nonnumeric character

Continuation lines

A line is a continuation line, when the last character (blanks included) of the preceding line was a '-'. The text starts with the first character.

Comment Lines

A line is a comment when the line is not a continuation line and the first character of the line is an asterisk ("*"). The program will automatically produce an output line with a "C" in the first column. Comment lines may not be continued.

LIBRARY FILE DESCRIPTION

Name: *GPAKDRAW

Contents: The object module of the GPAK console modelling test program.

Purpose: To allow the user to create, modify, and manipulate line drawings based on a ring type data structure, using the 2250 mod I display unit.

Usage: This file should be referenced by a $RUN command, with *GPAKDRAW+*GPAKLIB+*GPAPHLIB as the file to be loaded. Loading takes a minute or two, don't be alarmed at the delay. Storage requirements: 40-50 pages.

Logical I/O units referenced: None.

Examples: $RUN *GPAKDRAW+*GPAKLIB+*GRAPHLIB *SINK*

Description: The user at the 2250 interacts with *GPAKDRAW using the programmed function keyboard and lightpen. For detailed operating instructions, see the GPAK manual.

A summary of function key meanings follows:

0	Terminate execution of *GPAKDRAW.
1	Delete the current picture.
2	Get menu of old pictures.
3	Create new picture.
4	Delete element.
5	Degroup element.
8	Enter new point.
9	Draw new line from last point entered.
11	Move element.
12	Enter text element.
14	Draw new line.
15	Enter circular arc.
22	Get more model storage.
23	Temporarily erase element.
24	Redisplay all temporarily erased elements.
25	Scale or translate current picture.
28	Enter grouping mode.
29	Move group.
31	Haven't figured this one out yet. (REDSP2).

LIBRARY FILE DESCRIPTION

Name: *GPAKGRID

Contents: The object modules of the GPAK grid and plot test program.

Purpose: To demonstrate some of the grid display and curve plotting facilities available in GPAK.

Usage: This file should be referenced by a $RUN command, with *GPAKGRID+*GPAKLIB+*GRAPHLIB as the file to be loaded. Storage requirements: 20-25 pages.

Logical I/O units referenced:
6 - printout of diagnostic information. It is suggested that 6 be set to *DUMMY*

Example: $RUN *GPAKGRID+*GPAKLIB+*GRAPHLIB *SINK*;6=*DUMMY*

Description: The user at the 2250 console interacts with *GPAKGRID using the programmed function keyboard and lightpen. See the GPAK manual P.361 for detailed instructions. A summary of function key actions follows:

0	Set grid type to linear-linear
1	Set grid type to log x, lin y.
2	Set grid type to lin x, log y.
3	Set grid type to log-log.
4	1 x 1 grid
5	5 x 10 grid
6	20 x 20 grid
7	Full screen.
8	3/4 screen, centered.
9	3/4 screen, upper right position.
10	Set curve plot mode to points only.
11	Set curve plot mode to vectors only.
12	Set curve plot mode to points plus vectors.
13	Scissor to grid.
14	Use full screen
27	Fit new curve onto grid.
28	Generate new curve. Alternates between damped sinusoid and n-leafed rose. For rose, key in n, then hit 'end'.
29	Plot new curve (after 7-14 or 27).
30	Display new grid (after 0-9).
31	Terminate run.

Note: A 3/4 screen grid can be moved (within screen limits) by detecting it, then detecting new light pen position on the character raster which comes up. To detect, position light pen, push and hold down footswitch, and if nothing happens, hit any function key.

LIBRARY FILE DESCRIPTION

Name: *GPAKLIB

Contents: The object modules of the subroutines of GPAK, version II (plus necessary LIB and RIP cards to allow one pass, selective loading).

Purpose: Provides a variety of programming support for users of the 2250 mod I display unit, including attention handling, display management, and modelling. The modelling routines are independent, and may be used to create and manipulate a CORAL-like ring structure even for non-2250 programs.

Usage: These subroutines are used in conjunction with a user written graphics application program, which may be written in G-level FORTRAN IV, or in assembly language. *GPAKDRAW and *GPAKGRID contain the object modules of two such graphic application programs.

Reference: S/360 General Program Libarary, manual no. 360D-3.4.005: "GPAK: an online system 360 graphic data processing package with real time 2250 input and display." This manual contains detailed descriptions of all the subroutines, sample programs, operating instructions, and programming assistance.

Note: *GPAKLIB contains all the subroutines supplied in the GPAK version II release (except for GBDUMP, DUFFER, and PATCH - diagnostic subroutines not described in the GPAK manual). Some routines have been modified slightly from the release tape versions to make them compatible with MTS, but there should be no significant difference in behavior at the user level. Some of the GPAK subroutines call on a version of the OS express graphics routines and IBM written "PORS", which are available to MTS users in *GRAPHLIB.

LIBRARY FILE DESCRIPTION

Name: *GRAPHLIB

Contents: The object modules of part of the OS express graphic support package for the IBM 2250 mod I, modified (where necessary) to work under MTS. File includes necessary LIB and RIP cards to allow selective, one-pass loading.

Purpose: Contains open, I/O, and attention handling routines for the 2250, and order generation routines for certain kinds of plots, grids, labels, and arcs.

Usage: These routines are used by GPAK (see *GPAKLIB), and may (with caution - see note below) be called by any program needing the facilities of OS express graphics support.

Description: The following is a list of the subroutines contained in *GRAPHLIB, with a brief description of each:

GSTOR	store graphic orders
GCPRNT	display character information
GARC	generate arc with points
GSPLOT	plot with points
GCGRID	generate Cartesian grid
GVARC	generate arc with vectors
GSVPLT	plot with vectors
GLABEL	label a graph
GPGRID	polar grid with points
GPVGRID	polar grid with vectors
GOFFSG	check for off screen condition
ANLZ	analyze attention information
GIOCR	graphic I/O control routine
GOPEN	graphic open routine

Note: The MTS compatible version of OS express graphic support for the 2250 contained in *GRAPHLIB and *GRAPHMAC was designed to be minimally sufficient to assemble and run GPAK. Its general utility is untested, but it is thought that most usages of express graphics will transfer ok. The two important behavioral changes of the MTS version are 1) only one DCB may be open at a time (e.g. one cannot run both the 2250 and a printer as OS data sets simultaneously) and 2) attentions from the 2250 are handled on an interrupt rather than a polling basis (when ANLZ is called, it does not return until an interrupt has occurred).

Reference: IBM SRL C27-6921 "IBM System/360 Operating System: express

graphics programming services for the IBM 2250 Display Unit, Model I."

For detailed information on the OS versions of the open, attention handling, and I/O routines, see IBM PLM Y27-7113 "IBM System/360 Operating System: Graphics Access Method". This graphic access method has been revised and condensed into the three routines ANLZ, GOPEN, and GIOCR. The other subroutines in *GRAPHLIB are unchanged from the OS versions.

LIBRARY FILE DESCRIPTION

Name: *GRAPHMAC

Contents: A library of macros useful in programming for the IBM 2250 mod I display.

Usage: This file is in the form of a standard MTS macro library. Its name may be given as a macro library parameter to the assembler.

Example: $RUN *ASMBLR;SCARDS=PROGR 0=*SYSMAC 2=*GRAPHMAC PAR=CON

Description: This file contains a variety of graphic order generating macros, some graphic I/O macros which are needed by OS express graphics, and four special macros needed to assemble GPAK subroutines.

An exact list of the macro names can be obtained by listing the initial segment of this file (line numbers 1.0 thru 5.0).

Note: This file contains special versions of OPEN and STIMER. STIMER has been changed to a dummy, and OPEN is for graphic DCBs only.

Reference: IBM SRL C27-6921 "IBM System/360 Operating System : express graphics programming services for the IBM 2250 display unit, model I."

LIBRARY FILE DESCRIPTION

Name: *HEXLIST

Contents: The object module of the hexadecimal card lister

Purpose: To list cards (usually object cards) in hexadecimal

Usage: This module is invoked by the appropriate run command.

Logical I/O units referenced:

SCARDS	Where the file names are to come from
SPRINT	Where the listing is to be done

Examples: $RUN *HEXLIST
(SCARDS defaults to *SOURCE*; SPRINT to *SINK*)

Description: The output format is:

Line 1: The card number (advanced by 1 for each card read), the MTS file sequence number (line number), the card I.D. (columns 73-80 of the card), column 1 of the card (in hexadecimal), columns 2-4 in EBCDIC

Line 2: Columns 5-38 in hexadecimal

Line 3: Columns 39-72 in hexadecimal

*HEXLIST is interactive in the same way as *OBJSCAN is.

Sample output:

```
$RUN *HEXLIST; SCARDS=*SOURCE* SPRINT=*SINK*
#EXECUTION BEGINS
READY!
HXL(83.24,83.24)

CARD NO.  1,FILE SEQ. NO.=83.24      , CARD ID=HXL 0003, 02 , TXT
400000E0404000384040000147F0A0D44184 11F244420A2FED202A39BA309F321A1D4
A308DC01A1D4A2BED201A397A1D44110A3A058F0A41005EFF3E7A1D4A30CF3E7A1E23
```

LIBRARY FILE DESCRIPTION

Name: *IHC

Contents: Error descriptions for the error numbers produced by the Fortran library subroutines.

Usage: Line xxx of *IHC contains the description of error number IHCxxx. Hence if Fortran tells the user "IHC232I", then doing

```
$COPY  *IHC(232,232)
```

will bring forth an explanation of that error comment on his terminal.

LIBRARY FILE DESCRIPTION

Name: *LINPG

Contents: Object module of a calling program and the subroutine LINPG

Purpose: Provide a convenient way to solve linear programming problems.

Usage: The calling program is invoked by using a $RUN with *LINPG as the object file.

Logical I/O units referenced:

SCARDS	Where the data values are to be found
SPRINT	Where the results are put

Examples:
```
$RUN *LINPG
$RUN *LINPG; SCARDS=DATA
```

Description: Read the description of the subroutine LINPG. Values for M, N, M2, N2 using the IOH360 standard format 4I* are read in followed by values for the A array on the next card or line which are read in row by row using the IOH360 standard format 42F*. An example problem and the associated data:

maximize $X_1 + 2X_2 + 3X_3 - X_4$

subject to $x_1 + 2x_2 + 3x_3 = 15$

$2x_1 + x_2 + 5x_3 = 20$

$x_1 + 2x_2 + x_3 + x_4 = 10$

The data:
```
4 5 3 0
1.   2.   3.   0.   15.  2.  1.  5.  0.  20.  1.  2.  1.  1.
10.
-1.  -2.  -3.  1.  0.
```

The calling program interprets the switch returned by LINPG and prints the appropriate comments and results. See the description of the subroutine LINPG (in section MTS-253) for further details.

LIBRARY FILE DESCRIPTION

Name: *LISTVTOC

Contents: Object module to print out a list of files.

Usage: This file should be referenced by a $RUN with *LISTVTOC as the object file, and (optionally) the SCARDS source.

Logical I/O units referenced:

SCARDS (optional)	Gives serial numbers of disk packs whose files are to be listed
SPRINT	Catalog listing is given on the specified device

Examples: $RUN *LISTVTOC; SCARDS=RDRU SPRINT=PTR1

Description: *LISTVTOC lists on SPRINT the names of all the files allocated on the disk pack whose six-character volume serial number (name) is given in columns 1-6 of each input card (or line). An end-of-file causes a terminal exit, and requesting a non-existent or unmounted disk pack will generate an error message.

For input from SCARDS, if columns 7-10 are non-blank (if any one of them is non-blank) they are assumed to constitute a one to four character prefix and initiate a "selective catalog" print - i.e., only those files whose names begin with that prefix are printed.

If column 11 contains an X then the output is in hexadecimal; if it contains T then the files of the current user are listed by name only.

LIBRARY FILE DESCRIPTION

Name: *MOUNT

Contents: The object module of the volume mounting program.

Purpose: Allow users to mount private volumes and attach certain attributes to those volumes.

Usage: The program is invoked by a run command specifying *MOUNT as the file where the object cards are to be found.

Prototype:
```
$RUN *MOUNT; PAR=vol.i.d. [ON] devicetype,
     [PNAME=]*pseudodevicename*, [MODE=mode,]
     [SIZE=max.rec.size,]
     'volumename' [,comments to operator]
```

Examples:
```
$RUN *MOUNT;PAR=M000 ON 9TP,PNAME=*TEST*, SIZE=512,
     'TAPE TEST 1', RING IN

$RUN *MOUNT; PAR=M001 7TP *TEST2*, MODE=8CV,
     'TEST 2', RING OUT
```

Parameters: vol.i.d. is the 4 character rack number at the computing center in which the volume is stored.

devicetype is the type of device on which the volume is to be mounted. Currently 7TP and 9TP are supported.

pseudodevicename is the pseudo-device name by which the volume will be referred to in data accesses.

mode is the mode in which the volume was or is to be written.

size is the length in bytes of the longest possible record on that volume.

volumename is the identifying name associated with the volume. This name should be written upon a paper label and attached to the volume when it is first submitted to the computing center.

comments are specifications as to ring in or ring out, as well as any special directions to the operator.

Logical I/O Units Referenced: none.

Description: A comment will be printed on the operator's console requesting that he mount the specified volume on the specified device

type. If he is able to comply, the comment "OK" will be printed on the master sink device. If he is not able to mount the volume, a message from the operator explaining the problem will be printed on the master sink device. Volumes mounted in this way will automatically be removed when a second volume is mounted with the same pseudo-device name and when the user signs off.

The following rules describe the interaction between *MOUNT and *DISMOUNT.

1. If *MOUNT is run (either explicitly or by calling MOUNT during execution) with a pseudo-device name specified which is already active, then the device type must be the same as the previous type and all outstanding references to the pseudo-device name will be changed to refer to the new volume.
2. If *MOUNT is run with a pseudo-device name which has been previously used but for which *DISMOUNT has been run, then the new device type may be anything legal and any outstanding references to the old pseudo-device name are unchanged. In general a use of *DISMOUNT signifies the end of a particular use of a pseudo-device name and a use of *MOUNT for an active pseudo-device name signifies a new volume for a previous use.

The following example may clarify this. Suppose the following commands were issued during an MTS session:

```
$RUN *MOUNT; PAR=G000 ON 7TP PNAME=*TP* MY TAPE

$RUN -OBJ; SCARDS=*TP*
```

If during execution of the program in -OBJ, *MOUNT is called (presumably by using the macro or subroutine MOUNT) and the pseudo-device name *TP* is specified then the device type must be 7TP and a new reel will be mounted on the same tape drive, which SCARDS will still refer to. However if during execution of the program in -OBJ, *DISMOUNT is called for *TP* and then *MOUNT is called for it, the device type may be any legal type, SCARDS will still refer to the old tape, and any new use of the name *TP* (for example as a parameter to GETFD) will refer to the new volume. In this case the original tape reel will not actually be removed until execution is terminated.

LIBRARY FILE DESCRIPTION

Name: *NEWFORT

Contents: The object module for IEYFORT, the FORTRAN G control module, followed by the most recent IBM releases of the other six object modules that constitute OS FORTRAN G.

Purpose: Batch compilation of IBM/360 FORTRAN IV (refer to FORTRAN IV LANGUAGE, SRL C28-6515-4) source modules. The compiler can produce a source listing, storage allocation map, object listing, load module and/or object deck according to the options specified. The default options are: SOURCE, MAP, NOLIST, LOAD, NODECK, EBCDIC, NAME=MAIN, LINECNT=57.

Usage: The compiler is invoked by a RUN command with an objectFDname of *NEWFORT.

Logical I/O units referenced:

SCARDS	source modules
SPRINT	source, map and object listings
SPUNCH	card deck version of object module. The deck will have ID in columns 73 to 80 consisting of the first four characters of the program name followed by four digits--0001,0002,...
0	object module suitable for use as the objectFDname in a RUN command.

Scratch Files Referenced:

-T#	temporary file used by the compiler if the LOAD and/or DECK options are specified, containing information of no real value to anybody but FORTRAN.

Examples:

```
$RUN *NEWFORT; SCARDS=XSOURCE 0=-LOADMOD PAR=NOSOURCE,NOMAP
$RUN *NEWFORT; 0=-LOADMOD
(SCARDS and SPRINT default to *SOURCE* and *SINK*)
$RUN *NEWFORT; PAR=NOSOURCE,NOMAP,NOLOAD
```

Description: If the compiler runs to completion without finding any phase 1 errors (IEY001 to IEY018I,IEY032I) or if only phase 2

errors (IEY019I to IEY026I) are detected the compiler will terminate with the singular comment (produced by the system) EXECUTION TERMINATED. If a phase 1 error(s) was detected this comment will be preceded by the line:

***** FORTRAN RETURN CODE d

where d will be 0,4 or 8. Associated with each phase 1 error is a completion code of 0,4 or 8; it is the completion code corresponding to the most severe error that is given. There are five error conditions which may occur during compilation from which the compiler cannot recover. These will occasion the comment:

***** FORTRAN RETURN CODE d

where d=2,3,5,6 or 7. Subsequent to the printing of this comment the compiler will immediately call ERROR.

News of compiler changes, both internal and external, and compiler errors may be found in the system file *FORTRANNEWS. A copy may be most conveniently obtained by issuing a

$COPY *FORTRANNEWS TO *SINK*

command.

If parameters are to be given, the PAR= must be the last in the sequence of specifications as shown in the examples. A description of the options follow.

The FORTRAN IV language is described in "IBM SYSTEM/360 FORTRAN IV LANGUAGE", form C28-6515. The entire library as described in "IBM SYSTEM/360 OPERATING SYSTEM: FORTRAN IV, Library Subprograms", form C28-6596, is available and subroutines will be automatically loaded with the object module if required.

Note: The source file must be terminated either by an end-of-file or by a $ENDFILE command.

COMPILER OPTIONS

Options may be passed to the compiler through the PAR parameter in the RUN command, e.g., (default options are underlined).

```
                    SOURCE   MAP   LIST   DECK   LOAD   BCD
$RUN *NEWFORT; PAR=NOSOURCE,NOMAP,NOLIST,NODECK,NOLOAD,EBCDIC,
                    NAME=MAIN,LINECNT=57
```

These options may be specified in any order. If both the positive and

negative option are given, e.g., LIST,NOLIST, the NO- overrides the presence of the positive version. The last occurrence of the NAME and LINECNT keywords will be the effective ones.

SOURCE or NOSOURCE: The SOURCE option indicates that the source statements are to be written to the logical device SPRINT.

MAP or NOMAP: The MAP option specifies that a table of names, which appear in the object module, is to be written to the logical device SPRINT. The type and location of each name is given. The seven types distinguished are: COMMON variables, EQUIVALENCE variables, scalar variables, array variables, subprogram names, NAMELIST variables, and FORMAT statements.

LIST or NOLIST: The LIST option indicates that the object module listing is to be written to the logical device SPRINT. (The statements in the object module listing are in a pseudo assembly language format.)

DECK or NODECK: The DECK option indicates that the object module is to be written to the logical device SPUNCH in card deck form. The deck will have ID in columns 73 to 80 consisting of the first four characters of the program name followed by four digits -- 0001,0002,...

LOAD or NOLOAD: The LOAD option indicates the the object module is to be written to the logical device 0. This is a more compact version of the object module than that available on SPUNCH. The object module is written as a sequence of variable length records containing from 16 to 72 characters. No extraneous information is written.

BCD or EBCDIC: The BCD option indicates that the source statements are written in Binary Coded Decimal; EBCDIC indicates Extended Binary Coded Decimal Interchange Code, i.e., 026 versus 029 generation. Intermixing of BCD and EBCDIC in the source module is not allowed. The occurrence of the EBCDIC option overrides the presence of the BCD option. CAUTION: If the EBCDIC option is selected, statement numbers passed as arguments must be coded as &n. If the BCD option is selected, statement numbers passed as arguments must be coded as $n and the $ must not be used as an alphabetic character in the source module. (n represents the statement number)

NAME=xxxxxxxx: The NAME option specifies the name (xxxxxxxx) assigned to a module (main program only) by the programmer. If the name is not specified or the main program is not the first module in a compilation, the compiler assumes the name MAIN for the main program. The name of a subprogram is always specified in the SUBROUTINE or FUNCTION statement. The name appears in the source, map, and object module listings.

LINECNT=xx: The LINECNT option specifies the number of lines that will be written between page headings on the logical device SPRINT. If this parameter is not specified, the system default will be used.

The SOURCE, MAP and LIST options all cause output to the logical device SPRINT. This output is always written in the order just given, i.e., the source statements, the map, and finally the object module listing. The object module produced on SPUNCH (the DECK option) requires about 25% more file space than the object module produced on 0 (the LOAD option).

Note: The FORTRAN user's guide and a list of FORTRAN error messages will be found in section MTS-520.

LIBRARY FILE DESCRIPTION

Name: *OBJSCAN

Contents: The object module of the object-scan program

Purpose: To give an edited accounting of the contents of a object file.

Usage: The object-scan program is invoked by the appropriate run command specifying *OBJSCAN as the file where the object cards (for the OBJSCAN program) are found.

Description: The object-scan program reads an object file and gives an edited account of what is in that file. The program only looks at the second through fourth bytes of each image it obtains from the object file in order to make its decision; it will print on "SPRINT" the "BREAKS" when columns 2-4 change; for example it will tell where the break is between "ESD" and "TXT" object cards, between "RLD" and the "END" card, etc. The program will ask the name and beginning and ending line numbers of the file to be scanned; this request is made through "SERCOM"; the reply is made through "SCARDS". In addition to printing out on SPRINT where the breaks occur, the beginning and ending line number of a block of cards having the same characters in columns 2-4 are printed along with the id field of such line numbers (i.e. columns 73-80). At the beginning of the program and after each edit is finished, "READY" will be printed, thus inviting the user to enter another file or device.

Examples:

```
$RUN *OBJSCAN
    (SCARDS,SPRINT,SERCOM default to *SOURCE*, *SINK*,
*MSINK*, resp.)

$RUN *OBJSCAN SPRINT=PTRFIL
```

Example of usage. The information input by the user is underlined.

```
#$RUN *OBJSCAN
#EXECUTION BEGINS
READY!
*IEUASM
 "ESD"  IN LINE NO. 1      ASM 0001
 "TXT"  IN LINE NO. 2      ASM 0002 THRU LINE NO. 11     ASM 0011
 "RLD"  IN LINE NO. 12     ASM 0012
 "END"  IN LINE NO. 13     ASM 0013
 "LCS"  IN LINE NO. 14
 "LDT"  IN LINE NO. 15
READY!
```

LIBRARY FILE DESCRIPTION

Name: *OSMAC

Purpose: To enable OS users to assemble their programs in MTS. These programs must not be run under MTS.

Contents: The macro library from IBM's Operating System, Reference is "IBM System/360 Operating System: Supervisor and Data Management Macro-Instructions", form C28-6647.

Usage: This file should be specified for logical I/O unit 0 when running the assembler.

LIBRARY FILE DESCRIPTION

Name: *PAL8SS

Contents: Object module to convert PDP-8 binary paper tapes to PDP-8 load card format.

Usage: *PAL8SS is used to convert PDP-8 paper tape programs into a form suitable for use in the PDP-8 simulator or for use in PDP-8 systems with card readers.

Logical I/O units referenced:

SCARDS	Source of paper tape images
SPUNCH	Sink for binary load cards

Examples:
```
$RUN *PAL8SS; SCARDS=TAPESOURCE SPUNCH=CARDSINK PAR=10
$RUN *PAL8SS; SCARDS=*PAPERTAPERDR*@BIN
```

Parameters: The parameter represents the starting address in octal of the program. After all of the tape has been converted to cards a transfer card will be produced directing the card loader to transfer to the program at the assigned location. If no parameter is present 0 is assumed. For a complete description of the binary card formats see the 8ASS writeup (MTS-590). A description of the paper tape formats may be found in the appropriate DEC literature.

Note: Both the input and output FDnames refer to entities which must have binary attribute associated with them. Thus if TAPESOURCE is an intermediate file, $COPY *PAPERTAPERDR*@BIN TO TAPESOURCE would load it properly. Similarly $COPY CARDSINK TO *PUNCH*@BIN would punch the output. *PAL8SS both reads and punches @BIN, thus there is no need to explicitly attach the binary modifier to SCARDS and SPUNCH.

LIBRARY FILE DESCRIPTION

Name: *PIL

Contents: Object module of the PIL (Pitt Interpretive Language) interpreter.

Usage: This file should be referenced by a $RUN command. This is an interactive program designed to be used from terminals; use in batch is possible but of marginal utility.

Logical I/O units referenced:

SCARDS - input to PIL

SPRINT - output from PIL

Example:
```
$RUN *PIL
     (SCARDS and SPRINT default to *SOURCE* and *SINK*)
```

Description: See the PIL writeup in section MTS-550 of this manual.

LIBRARY FILE DESCRIPTION

Name: *PLOT

Contents: Object module to produce plots of points.

Usage: This file should be referenced by a $RUN with *PLOT as the object file.

Logical I/O units referenced:

SCARDS	Where the points and the parameter are to be found.
SPRINT	Where the plot is put.

Examples:

```
$RUN *PLOT;SCARDS=*SOURCE* PAR=NSBH=9,-
LABEL='EXPERIMENT 1'

$RUN *PLOT;SCARDS=DATAFILE SPRINT=PTRFILE -
PAR=NSBH=9,NVL=11,LOG=1,LABEL='COST OF LIVING'
```

Description: The user is asked to enter the number of points he wishes to plot followed by the points themselves. Each point is given by its abscissa and ordinate. The points (1,2.5), (2.1,4.6) would be entered as 1 2.5 2.1 4.6. After the plot has been produced, the user is asked whether he wishes to plot again. By specifying new parameters beginning in column 1, a new plot of the same points will be produced. If NEW is included in the specifications, the user will be asked to supply a new set of points which will then be plotted. Execution will be terminated by putting an S in column 1.

The parameters are:

NHL=nn	The number of horizontal lines, nn≥2
NSBH=nn	The number of spaces between horizontal lines, nn≥1.
NVL=nn	The number of vertical lines, nn≥2
NSBV=nn	The number of spaces between vertical lines, nn≥1.
LABEL='label'	A vertical label centered along the left side of the graph.

NEW	New points are to be used in the next plot, or if in function mode, a new file name will be requested.
LOG=nn	nn=0 Plot is linear on both axes nn=1 The Y axis is scaled logarithmically nn=2 The X axis is scaled logarithmically nn=3 The X and Y axes are scaled logarithmically
F	Instead of supplying points, the user is asked to specify the name of a file containing a FORTRAN compatible function. This function will be called to produce points in the interval [XL,XR]. Values for XL and XR should be specified at the same time that F is given. The object records in the file must end with an LDT record, with the function entry point specified. (See the Dynamic Loader writeup)
XL=x	The left boundary value of the plot if F is used.
XR=x	The right boundary value of the plot if F is used.
NOF	The function feature is turned off.
CHAR=c	The plotting char is set to c
Y	If F is set, the user sets up values for YH and YL
YH=y	The highest value of the function that is plotted.
YL=y	The lowest value of the function that is plotted.
R	The X and Y coordinates are interchanged.
NOR	The R feature is turned off.
NOY	The entire function is plotted.

OMIT=nn

This will cause the deletion of any of the following parts of the graph depending on the value of nn.

1 Delete X grid values

2 Delete Y grid values

4 Delete the last graph line

Several parts may be deleted by setting nn equal to the sum of the numbers corresponding to the parts. For example, nn=7 would cause all three parts to be omitted. If one or more functions are to be restored, nn should be set to the negative sum of the numbers. For example, nn=-3 would restore the X and Y grid values.

The default cases for the parameters are:

```
NHL=6,NSBH=1,NVL=6,NSBV=9,LABEL=' ',LOG=0,NOF,CHAR=*,NOY
```

Example of usage of *PLOT:

In the following example, characters that the user typed in have been underlined to make it easier to follow.

```
#$RUN *PLOT
#EXECUTION BEGINS
 HOW MANY POINTS ARE TO BE PLOTTED?
 3
 ENTER ABSCISSA AND ORDINATE OF EACH POINT
 1. 1. 2. 2. 3. 3.
      3.000 +---------+---------+---------+---------+---------+---------*
            I         I         I         I         I         I         I
      2.600 +---------+---------+---------+---------+---------+---------+
            I         I         I         I         I         I         I
      2.200 +---------+---------+---------+---------+---------+---------+
            I         I         I    *    I         I         I         I
      1.800 +---------+---------+---------+---------+---------+---------+
            I         I         I         I         I         I         I
      1.400 +---------+---------+---------+---------+---------+---------+
            I         I         I         I         I         I         I
      1.000 *---------+---------+---------+---------+---------+---------+
          1.000     1.400     1.800     2.200     2.600     3.000
 IF YOU WISH TO PLOT AGAIN WITH DIFFERENT PARAMETERS,
 PUT THEM IN COLUMN 1
 OTHERWISE PUT AN S IN COLUMN 1
 NHL=2,NVL=2,LABEL='L',NSBV=4
      3.000 +----*
 L          I  * I
      1.000 *----+
         1.0003.000
 PUT PARS OR S IN COLUMN 1
 S
#EXECUTION TERMINATED
```

LIBRARY FILE DESCRIPTION

Name: *ROSSPRINT

Contents: Object module to print the upper/lower case print tapes (ROSS print tapes) produced by TEXT90.

Usage: This file should be referenced by a $RUN with *ROSSPRINT as the object file, SPRINT as output, and SCARDS as the input tape.

Logical I/O units referenced:

SCARDS	The input tape (ROSS print tape)
SPRINT	The writeup which is output is produced on the specified device.

Examples:
```
$RUN *ROSSPRINT; SCARDS=*TAPE* SPRINT=PTR1 PAR=UC,DBL
$RUN *ROSSPRINT; SCARDS=T0C1 SPRINT=PTR3 PAR=NOCTL
$RUN *ROSSPRINT; SPRINT=OFILE SCARDS=*IN*
```

Note: Output put in a file should be later printed with application of the machine carriage control modifier. For example:

```
$COPY  OFILE  TO  PTR1@MCC
```

Description: This program prints the special 1401 ROSS output tape (also called the upper/lower case print tape) produced by TEXT90 as modified for U. of M. use. Specifically, the tape is written in high (556 BPI) rather than low density, and the self-loading 1401 program and trailing tape mark are omitted. [i.e., the data is the first file on the tape, not the second.] The output can be printed single or double spaced, with or without the TEXT90 control fields (for editing purposes), can include all pages in the document or only those pages which were changed during the TEXT90 run, and can print with full character set or can map lower case letters into upper case letters so a test print can be made on a printer without installing the TN print train. These options are controlled by the use of parameters in the PAR= field of the $RUN command.

For normal usage (unless UC has been specified) the TN print train should be installed and the UCS buffer loaded appropriately (via the TN program in UMMPS).

The character set available on the TN print train differs slightly from that used on the special 1401 chain originally used to print TEXT90 ROSS tapes. Specifically, left arrow, up arrow, carat, and subscript 1, 2, 3, and n, are not available. Cent sign, square block, logical not, pillow, and superscript plus, minus, left parenthesis and right parenthesis are available on the TN train in their place. Accordingly, the following substitutions for graphics should be made to the special-character notation in using TEXT90:

1401 ROSS Chain	TN train	TEXT90 notation
↑ (up arrow)	¢	/C
← (left arrow)	▪ (square block)	/L
^ (carat)	¬ (logical not)	//C
subscript n	superscript right parenthesis	//N
subscript 1	superscript plus	//1
subscript 2	¤ (pillow)	//2
subscript 3	superscript left parenthesis	//3

The following sub-fields are valid in the PAR= field of the $RUN command. Illegal sub-fields are ignored. Default assumptions for each pair are underlined. All are optional. Order of occurrence does not matter.

SNGL	Output will be single spaced.
DBL	Output will be doubled spaced
CTL	TEXT90 control fields and exception listings are printed.
NOCTL	TEXT90 control fields and exception listings are not printed (used for making final copy for reproduction or distribution)
ALL	All pages in the document are printed.
CHNGD	Only those pages altered in the TEXT90 run are printed.
UC	Maps all lower case letters to upper case letters so a non-TN train may be used. (is normally off)

LIBRARY FILE DESCRIPTION

Name: *SDS

Contents: The object module of the Symbolic Debugging System program.

Purpose: To aid in the development and checkout of assembly language programs.

Usage: SDS is invoked with an appropriate RUN command.

Logical I/O units referenced:

SCARDS	The source from which SDS reads its commands.
SPRINT	SDS messages and diagnostics.

Examples:
```
$RUN *SDS
$RUN *SDS; SCARDS=COMS SPRINT=DIAGS
```

SDS USER'S GUIDE

SDS is a program checkout system which allows the assembly language programmer to debug his programs conversationally (i.e., from a remote terminal.) Using the SDS command language, the programmer initiates execution of his program and interrupts it at specified break-points, where instructions and data may be displayed and modified.

I. Program Preparation

SDS processes object programs produced by the MTS Assembler. The specification, PAR=TEST must be set when a program to be tested by SDS is assembled. This specification causes the assembler to include a symbol table with the object deck it produces, thus allowing the user and SDS to reference the program symbolically. The Loader will unconditionally ignore the symbol table when it goes to load the program.

II. Invoking SDS

SDS is invoked with a $RUN *SDS command. When execution begins, SDS types:

```
+ENTER OBJECT FILE NAME.
```

The user responds by entering the name of the file (or concatenated file name) containing the object program to be tested. If multiple control sections are loaded as one or more assemblies, the first control section loaded will become the current CSECT (see $CSECT command.) When the user's object program has been loaded, SDS types:

```
+READY
```

at which point the user may type in his first command. In general, SDS signals its readiness to accept a command by typing the prefix character, "+", in column one. This same prefix character precedes all SDS messages and diagnostics.

III. Command Language Specifications

The user communicates with SDS through the SDS command language by entering command lines from his terminal. A command is a request that SDS do something with the program to be tested: for instance, $DISPLAY displays instructions and data on the user's terminal; $MODIFY alters the contents of memory locations or the arithmetic registers; $BREAK enters "break-points" into the user's program in order to return control to the user at strategic points in his program. Every SDS command begins with a dollar sign followed by three or more alphabetic characters. A list of one or more parameters usually follow each command.

A parameter is a means of referencing specific locations (loaded addresses) in the program being tested with SDS. The command language includes two types of parameters: simple parameters and block parameters. A simple parameter is any of the following:

(1) a label in the label field of an instruction, DC or DS statement; a label created by the $LABEL command.

(2) a relative address in the form of an unsigned hexadecimal constant.

(3) a label or a relative address plus or minus a hexadecimal constant called the displacement. The displacement is the number of bytes following or preceding the location specified by the label or relative address.

(4) a label or a relative address immediately followed by a parenthesized subscript, i, representing the ith assembled element beyond the label or relative address. The subscript is assumed to be an unsigned decimal integer.

(5) a subscripted label or relative address followed by a displacement.

A block parameter consists of a simple parameter followed by three periods (an ellipsis) followed by a second simple parameter. The loaded address corresponding to the first parameter must be strictly less than the loaded address corresponding to the second parameter.

It is frequently necessary to distinguish between a label and a hexadecimal constant. For instance, the string of characters, "ABC", could be the label, ABC or the constant, ABC (base 16). When the latter interpretation is intended, the surfix, "@X", must be appended to the constant. In the previous example, we would have "ABC@X" for the hex constant. This explicitness is needed specifically to distinguish between labels and relative addresses appearing in SDS parameters; SDS assumes all displacements are hexadecimal and all subscripts are decimal integers. If the first character of a relative address is numeric (0,1,...,9) "@X" need not be specified.

The following assembly language program will illustrate the use of SDS parameters:

```
000000   PROG     START 0
                  USING *,15
000000            L     1,ALPHA
000004   AD       A     1,C
000008            ST    1,ALPHA
00000C            L     10,VSYS
000010            BR    10
000014   ALPHA    DC    F'10'
000018   VSYS     DC    V(SYSTEM)
00001C   C        DC    F'20'
000020   GAMMA    DS    20F
000070   DELTA    DC    2P'1,22,3333,'
                  END
```

Following is a list of SDS parameters and their corresponding relative addresses:

PARAMETER	ADDRESS
AD	000004
VSYS	000018
C	00001C
70	000070
C@X	00000C
AD+4	000008
VSYS-10	000008
VSYS-A	00000E
20+4	000024
C+C	000028
C@X+C	000018
ALPHA(1)	000018
10(2)	000018
PROG(7)	000018
DELTA(3)	000076
70(2)	000073
C(1)	000020
C@X(1)	000010
GAMMA(1)	000023

```
PROG(1)+4       000004
C@X(1)-4        00000C
ALPHA...DELTA 000014...000070
AD+4...C@X(1) 000008...000010
```

IV. Command Definitions

Following is a list of commands processed by SDS. In general, only the first three characters after the dollar sign need be typed. The structure of the the parameter list for each command in presented in standard MTS notation.

$ATTRIBUTES [simple-parameter] [,simple-parameter] ...

The attributes for each simple parameter are listed. These attributes include: relative address, loaded address, type, length, and (if different from one) scale factor and duplication factor. The one-character codes for "type" are defined in the discussion of the $DISPLAY command.

Example: $ATTRIBUTE GAMMA would print out

```
RA=000020  LA=101698  TYPE=F  LENGTH=4  DUP=20
```

$BREAK [simple-parameter] [,simple-parameter] ...

At each location specified by the parameter list, the instruction is saved and a "break-point" is inserted into the user's program. When control reaches a break-point during execution of the user's program, control is returned to SDS which prompts the user for his next command. The program may be restarted by issuing a $RESTART command which executes the saved instruction and transfers control to the user's program where normal sequencing resumes.

```
        ┌          ┐
        |csect-name|
$CSECT  |    *     |
        | integer  |
        └          ┘
```

When a program consisting of one or more control sections is loaded, the first one loaded becomes the current CSECT and the remaining control sections are opened for referencing. If SDS cannot locate a given label in the current CSECT, the remaining CSECT's are searched (in the order they were loaded) for a possible match. If SDS locates the label in the current CSECT, SDS will search through the remaining open CSECT's to make sure the

label is not multiply defined. A warning message is printed for each multiply-defined label thus located and the first definition is used.

The $CSECT command provides a means of specifying a new current CSECT. The control section named by csect-name becomes the current CSECT and the remaining control sections are closed; that is, checking for multiply-defined labels is suppressed and, if a label cannot be found in the current CSECT, no further searching through the remaining control sections takes place.

If the command $CSECT * is entered, the first control section loaded becomes the current CSECT and the remaining control sections are opened for referencing.

If one or more private control sections has been loaded, the command $CSECT i makes the ith private control section loaded the current CSECT and closes the remaining control sections.

Examples: $CSECT PROG1
$CSECT *
$CSECT 2

```
          ┌                 ┐ ┌                    ┐
          |simple-parameter | | simple-parameter    |
$DISPLAY  |block-parameter  | |,block-parameter    | •••
          └                 ┘ └                    ┘
```

Each parameter (or element in the range of a block parameter) is converted according to its type and length, and printed out along with a one-character code which indicates the type. These codes are defined as follows:

CODE	TYPE
A	address constant
B	binary
C	character
E	floating point
F	fixed point
I	instruction format
N	alignment exception - hex assumed
P	packed
S	S-type address constant
V	V-type address constant
W	channel command word format
X	hexadecimal
Y	Y-type address constant
Z	zoned decimal

If a parameter to $DISPLAY specifies an address which is incorrectly aligned with respect to the parameter's type, (e.g., half-word alignment for a full-word integer) the contents of the byte specified by the parameter, and the contents of the next seven bytes, are printed in hexadecimal.

The following conventions will be used when $DISPLAY attempts to display "illegal" data:

(1) character conversion: if a hex code is encountered for which there exists no graphic, a question mark will be substituted for that code.

(2) instruction format : if the op-code portion of an instruction is not a defined operation, the op-code is printed as two hex digits sandwiched between asterisks; the length of such an instruction is assumed to be two.

(3) packed format : if the length of the number exceeds 16, the number is dumped in hexadecimal.

(4) zoned format : same as (3).

To display the 16 general registers, the predefined parameter, "GRS" is used. To display the ith general register, the parameter, "GRi" is used, where i is the register number, ranging from zero to 15.

To display the four floating point registers, the predefined parameter, "FRS" is used. To display the jth floating point register, the parameter, "FRj" is used, where is j is the register number, ranging from zero to seven.

To display a location labelled "GRS" or"FRS", or "GRi" or "FRj", where i and j are as defined above, append the override, "@q" to each symbol. Thus, to display the location labelled "GR10, enter the command, $DISPLAY GR10@Q; to display general register 10, enter $DISPLAY GR10.

```
Examples: $DISPLAY ALPHA, GAMMA(1)...GAMMA(20)
          $DISPLAY PROG+20, GRS, FR2,C(1)-3
```

$GOTO [simple-parameter]

Control is transferred to the location in the user's program specified by the parameter. The parameter must specify a half-word aligned address.

```
          ┌                    ┐ ┌                     ┐
          |simple-parameter    | | simple-parameter    |
$HEXDISPLAY |block-parameter     | |,block-parameter     | ...
          └                    ┘ └                     ┘
```

Each parameter is printed out in hexadecimal in a format identical to the format used by SDUMP.

```
        ┌                                    ┐
        |                  {decimal-constant}|
$IGNORE |simple-parameter {simple-parameter}|
        └                                    ┘

              ┌                                    ┐
              |                  {decimal-constant}|
              |,simple-parameter {simple-parameter}| ...
              └                                    ┘
```

Ordinarily, when control reaches a location in the user's program containing a break-point, execution is interrupted and control returns to SDS. The $IGNORE command provides a means of suppressing this interruption each time control reaches the break-point location, for a total of N times, where N is either a decimal constant or the contents (presumably integer) of the location specified by the simple-parameter. The N+1st time control reaches the break-point, the interruption takes place as usual.

For example, assume a break-point exists at location AD in the sample program in section III. To suppress the interruption 20 times, either of the following commands could be used:

```
$IGNORE AD 20    or    $IGNORE AD C
```

```
$LABEL [simple-parameter label]
       [,simple-parameter label] ...
```

The location specified by simple-parameter is assgned the symbol specified by label. Thereafter, the location may be referred to by that symbol. label is a string of one through eight alphabetic or numeric characters, the first of which is alphabetic.

$MAP

A Loader-like map is printed, listing each control section in the user's program.

```
        ┌                                ┐
        |simple-parameter hex-constant   |
$MODIFY |block-parameter hex-constant     |
        └                                ┘

             ┌                                ┐
             | simple-parameter hex-constant  |
             |,block-parameter hex-constant   | ...
             └                                ┘
```

If simple-parameter is specified, its corresponding hex-constant is copied into the location specified by the parameter. If block-parameter is specified, hex-constant is sprayed through the region delimited by the block parameter. In any case, the length of hex-constant must be no greater than the length attribute of any location into which the constant is being entered. The hex constant will be right-justified and padded with leading zeros if necessary.

To modify the contents of a general or floating point register, the naming conventions as specified in the description of $DISPLAY must be used.

$RESTART

This command returns control to the next instruction to be executed after a program interrupt or a $BREAK-produced interrupt has occurred. in the latter case, the saved instruction originally at the break-point location is executed and execution proceeds as usual.

$RESTORE [simple-parameter] [,simple-parameter] ...

If no parameters are specified, the last break-point entered via the $BREAK command is removed from the user's program and the original instruction is restored. If one or more parameters are specified, the break-point(s) at each location are removed and the original operation restored.

$RUN

Control is transferred to the entry point of the user's program.

$STOPTEST

End of testing. Control is returned to MTS.

V. Parameter Overrides

It is occasionally useful to alter the implied or assumed attributes of an SDS parameter. For instance, it might be necessary to display a hexadecimal constant in EBCDIC format. SDS provides a means of temporarily overriding the attributes of a parameter by explicitly specifying those attributes. Each override consists of an ampersand (@) followed by a one-character override code, followed (if necessary) by a constant. When overriding the attributes of a parameter, the override code is appended directly to the parameter. If necessary, more than one override may be appended. Attributes may not be used with block parameters.

SDS also provides a means of specifying information about the constituents of a parameter. For instance, it might be useful to express a displacement as a decimal rather than a hexadecimal constant; or a single hex constant might be an absolute (virtual) address rather than a relative address. In this case, override codes specifying these properties can be appended to that element of a parameter (i.e., label, displacement,subscript, etc.) which is being modified.

Following is a list of override codes and their definitions.

CODE	DEFINITION
@A	treat as an absolute (virtual) address.
@D	force decimal conversion
@L=i	set the length attribute to i, where i is an unsigned decimal integer.
@Q	treat as a label
@S=i	set the scale factor to i, where i is a signed or unsigned decimal integer.
@T=code	Set the type attribute to code, where code is any of the single-character type-codes defined in the explanation of the $DISPLAY command.
@X	force hexadecimal conversion

Examples:

Specify the tenth byte beyond the location labelled "AD":

```
AD+A    or    AD+10@D
```

Force a length of 20 on GAMMA:

```
GAMMA@L=20
```

Force character conversion and a length of two on delta:

```
DELTA@T=C@L=2
```

Force hex conversion for the contents of location 00000C:

```
C@X@T=X
```

Specify absolute (virtual) address 130108 (base 16):

```
130108@A
```

LIBRARY FILE DESCRIPTION

Name: *SNOBOL4

Contents: The object module of the SNOBOL4 interpreter.

Usage: The SNOBOL4 interpreter is invoked by means of a RUN command. Since SNOBOL4 currently does all I/O through the FORTRAN IV I/O routines, all input and output is done through logical I/O units 1 to 9 See the Fortran Users Guide (MTS-520) for details on I/O.

Logical I/O units referenced:

5 Source for the SNOBOL4 program to be translated, immediately followed by:
Data read via default "INPUT" string.

6 Output from compilation of the SNOBOL4 program and output via the default "OUTPUT" string.

7 Output via the default "PUNCH" string.

Examples
```
$RUN *SNOBOL4; 5=*SOURCE* 6=*SINK*
$RUN *SNOBOL4; 5=SNOPROG+*SOURCE*
  6=*DUMMY*(1,20)+*SINK* 7=*PUNCH*
```

Description: See the SNOBOL4 language description in section MTS-560 (volume II) of the manual.

LIBRARY FILE DESCRIPTION

Name: *SQUASH

Contents: The object module of the Card-Squashing Program.

Purpose: To squash object cards to as few records as possible.

Usage: The Card-Squashing Program is invoked by the appropriate RUN command specifying *SQUASH as the file were the object cards of the Card-Squasher may be found.

Logical I/O Units Referenced:

SCARDS - the file or device which contains the object cards which are to be squashed.

SPUNCH - the file or device on which squashed object records are to be placed.

SERCOM - the file or device where error comments are to be placed; error comments do NOT cause ERROR RETURN (i.e. they are not fatal).

Examples:

```
$RUN *SQUASH; SCARDS=-LIB SPUNCH=MYLIBRARY

$RUN *SQUASH; SPUNCH=-PUNCH PAR=IDS
    (SCARDS defaults to *SOURCE*)
```

Description: The card-squashing program compresses object cards which may be produced by a system translator (TXT,ESD,RLD, and END cards) and those cards which may be hand-produced (LIB,RIP,REP,DEF,LDT, and LCS cards). The input to the card squashing program consists of 80 (or less) byte card images---bytes 73-80 contain card ID; the output consists of compressed "card" images which may be up to 255 bytes in length. RLD and TXT input cards are formed into larger images for output so that, in effect, there will be fewer of these types of images in the output (compared with the input). The other types of card images except for REP are trimmed from column 72 backwards. REP cards are trimmed starting at the first blank character in the data field. The identification of the input images may be placed in the output images by using the parameter 'IDS' ---this is done on a one-for-one basis for all cards except TXT and RLD cards; the IDS of the TXT and RLD cards are the IDS of the first input record used in the compressed output image.

LIBRARY FILE DESCRIPTION

Name: *SSP

Contents: The MTS FORTRAN produced object modules of the most recent modification level of the most recent version of the Scientific Subroutine Package. A list of SSP entry points is included in section MTS-253.

Usage: These subroutines are thoroughly documented in the IBM publication System/360 Scientific Subroutine Package, (360A-CM-03X) Version II, Programmer's Manual (H20-0205) They are made available to a program by concatenating *SSP with the file or device containing the object program.

Example: $RUN -OBJ+*SSP *SINK*; 5=*SOURCE* 6=*SINK*

LIBRARY FILE DESCRIPTION

Name: *STATUS

Contents: The object module of the user space-time status program.

Purpose: Print information regarding the user's elapsed time, core time, CPU time, disk space, and disk storage time.

Usage: The space-time status program is invoked by an appropriate RUN command specifying *STATUS as the file where the object cards are found.

Logical I/O units referenced:

SPRINT Listing of the space-time information.

Examples:
```
$RUN *STATUS
$RUN *STATUS;SPRINT=FILE
```

Description: *STATUS lists on SPRINT the following information for the user. The maximum, used, and remaining figures are given for cumulative elapsed time in minutes, cumulative core usage in page hours, cumulative CPU time in minutes, and current disk space in pages. The amount used is from the last time the cumulative sums were reset (e.g. the end of a billing period) to the last signoff before the current signon. In addition, the cumulative disk storage in page-hours is given from the last time the cumulative sums were reset to the last time a disk file (including temporaries) was created, destroyed, or changed extents.

LIBRARY FILE DESCRIPTION

Name: *SYMBOLS

Contents: Object module to print a listing of all external symbols within the MTS system.

Usage: This file should be referenced by a $RUN with *SYMBOLS as the object file, SPRINT as output, and SCARDS as the location of a control record.

Logical I/O units referenced:

SPRINT	The listing which is output is produced on the specified file or device
SCARDS	A control record is read from the device or file specified.
Note:	*SYMBOLS(0) is a valid control record.

Example:
```
$RUN *SYMBOLS; SCARDS=*SYMBOLS(0)
     (SPRINT default to *SINK*)
```

Description: This program produces the listing of MTS external symbols which is reproduced in section 253 of this MTS manual. The listing is produced by searching the external symbol dictionaries of the system and the library. Hence it is current as of the instant it is run.

The program has options to generate printed output or a deck in TEXT90 format. It also will suppress privileged system symbols if desired. These options are set by a control record which the program expects on the SCARDS file or device. The format of this control record is not provided in this writeup.

There is a control record in the file *SYMBOLS at line number 0. This record will cause the program to produce printed output with privileged symbols suppressed. Hence a user may obtain a listing by the command:

```
$RUN *SYMBOLS;SCARDS=*SYMBOLS(0)
```

LIBRARY FILE DESCRIPTION

Name: *TABEDIT

Contents: The object module of the tab editing program.

Purpose: To simulate the TAB key on remote terminals.

Usage: The program is invoked by an appropriate RUN command, specifying an input file, an output file and a parameter list, containing a character to represent TAB and a list of tab settings.

Logical I/O units referenced:

SCARDS The input file to be edited.

SPRINT The output file containing the edited lines.

Examples:

```
$RUN *TABEDIT; SCARDS=*SOURCE* SPRINT=ACTIVE
     PAR=10,16,35,72

$RUN *TABEDIT; SCARDS=AA SPRINT=BB PAR=/,5,10,15
```

Description: The user defines the TAB character and tab settings in the PAR list. If the first character in the PAR list is non-numeric, the character will be used as the TAB code, and all occurrences of that character in the input file will be treated as a tabulator key. If the first character in the PAR list is numeric, the normal code for TAB will be used.

The tab stops are entered into the PAR list as column numbers. For instance,

PAR=/,5,10,15

specifies "/" as the TAB character, and tab stops at column 5, 10 and 15. An occurrence of "/" in an input line will cause blanks to be inserted up to the next tab stop. If the line pointer is beyond the last tab stop (e.g., in the previous example, column 15 or beyond) one blank will be inserted in the output line for subsequent occurrences of "/".

If a TAB character other than normal TAB is used, occurrences of normal TAB in an input line will not cause tabulation. All occurrences of the TAB character are deleted from output lines.

LIBRARY FILE DESCRIPTION

Name: *UMIST

Contents: Object module for UMIST interpreter.

Purpose: To process UMIST procedures.

Usage: UMIST is an interactive text-processing language. It interprets strings of characters accepted one at a time from the input device and prints the value of each string after processing.

Logical I/O units referenced:
The initial settings for input and output devices are the SCARDS and SPRINT logical devices. Any file or device name may later be designated as the current input or output unit.

Examples: UMIST may be invoked by the MTS command

```
$RUN  *UMIST
```

Description: See the UMIST writeup (section MTS-570).

LIBRARY FILE DESCRIPTION

Name: *UPDATE

Purpose: This program will copy tapes (or files) containing card images, making insertions and deletions, as well as blocking and unblocking.

Logical I/O Units Referenced:
SPRINT - printed output
SERCOM - error messages
SPUNCH - output from %PUNCH

Commands and insertions are expected to come from the source stream (*SOURCE*). If another source of commands is wished, its FDname should be specified following the "PAR=" on the $RUN command.

Examples:
```
$RUN  *UPDATE
$RUN  *UPDATE; PAR=PIL.UPDATE
```

Usage: The update input tape must consist of 80 column card images which may be blocked to any factor desired. The blocking factor, if greater than 1, must be stated on the %INPUT command. The update output tape will consist of 80 column card images blocked as specified (except for the last record, and other records which may be truncated by a %CLOSE command. [Space for the specified input and output buffering is obtained dynamically when %INPUT and %OUTPUT command are encountered; released when %CLOSE is encountered.]

All commands take the following form: column 1 must contain a percent-sign ("%") which must be immediately followed by the command (only the first three letters need be given, and they must be upper-case). Parameters for the command are separated from the command and from each other by one or more blanks (or commas, which are treated identically with blanks). Lines which are not recognized as commands are treated as insertion lines and are copied immediately to the update output tape.

Tape positioning and reading across a file mark while the tape is still open is considered an error. Therefore %CLOSE commands should precede positioning and the update should finish copying of a file with %BEFORE FILEMARK, not %AFTER FILEMARK.

Tape Attachment and Manipulation Commands

```
%INPUT   INTAPE   [N]
```

INTAPE is the pseudo-device name (or FDname) of the file or device to be established as the update input tape. N is an integer specifying the blocking factor of that tape. If omitted, a blocking factor of 1 card/record (i.e., unblocked) is assumed. This command causes the tape to be opened.

Example: %INPUT *IN* 50

%OUTPUT OUTTAPE [N]
OUTTAPE is the pseudo-device name (or FDname) of the file or device to be established as the update output tape. N is an integer specifying the blocking factor desired on that tape. If omitted a value of 1 (unblocked) is assumed.

Example: %OUTPUT *OUT* 20
%OUTPUT FILE1 1

%REWIND T
Tape T is rewound. T must not currently be open as input or output.

Example: %REWIND *OUT*

%RUN T
%UNLOAD T
Tape T is rewound and unloaded. T must not currently be open as input or output.

Example: %RUN *OUT*

%FSF T [N]
Tape T is spaced forward N files. If N is omitted a value of 1 is assumed. T must not currently be open as input or output.

Example: %FSF *IN* 3

%BSF T [N]
Tape T is spaced backwards N files. If N is omitted a value of 1 is assumed. T must not currently be open as input or output.

Example: %BSF *IN*

%WTM T [N]
%WEF T [N]
%EOF T [N]
Tape T has N tape marks (end-of-file marks) written on it. If N is missing a value of 1 is assumed. T must not currently be open as input or output.

Example: %WTM *OUT*

%CLOSE [T]
Tape T is closed. If the update output tape, the last buffer (possibly truncated) is written out. If T is omitted, the update output tape is assumed.

Update Feature Control Commands

%NEWID H
Cards written onto the update output tape following this command will have new I.D.'s (columns 73-80). The first card written will have the I.D. specified in this command. Succeeding cards will have I.D. incremented in steps of one. The I.D. given in this command should consist of 8 characters - blanks and commas may not be included. Only the numeric portion of the I.D. is incremented.

Example: %NEWID PIL00001

%OLDID
Suspends the re-I.D.ing of the cards as described under %NEWID.

%LIST ON
Starts listing of deleted and inserted cards. (Initially on)
OFF
Stops listing of deleted and inserted cards [This output goes onto *SINK*]

%PUNCH ON

Starts putting all card images sent to the update output tape on SPUNCH (presumably for punching).
OFF
Turns off the "punching" described above. (Initially off)

Card Location, Copying and Deletion Commands
In the execution of these commands, the 360 collating sequence is used for comparisons.

%AFTER ID
Copies all cards having I.D.'s less than or equal to ID from the update input tape to the update output tape
N
Copies the next N cards from the update input tape to the update output tape.

Examples: %AFTER PIL03789
%AFTER 2

%BEFORE ID

Copies all cards having I.D.'s less than ID from the update input tape to the update output tape

FILEMARK
FILEMK

Copies the rest of the file. It leaves the tape positioned after the file mark.

Examples: %BEFORE PIL07892
%BEFORE FILEMARK

%DELETE ID

Copies all cards having I.D.'s less than ID from update input to update output, then deletes the card (or cards) having I.D. of ID (if any).

N

Deletes the next N cards on the update input tape

ID1 ID2

Copies all cards having I.D's less than ID1 from update input to update output, then deletes all cards having I.D.'s ID1 through ID2 inclusive from the input.

ID1 N

Copies all cards having I.D.'s less than ID1 from update input to update output, then deletes the next N cards on the input tape.

* ID2

Deletes all cards on the update input tape from the current position up through ID2.

Examples %DELETE PIL00016
%DELETE 2
%DELETE PIL00378,PIL00379
%DELETE PIL00378,2
%DELETE * PIL00736

%FIND ID

The update input tape is searched for a card with I.D. equal to ID. Order of the I.D.'s is ignored. Card passed over are not copied to the update output tape.

Example: %FIND QQSV0395

Return Command

%END

This command or an end-of-file encountered in the command stream causes execution of the update program to terminate. All buffers are closed.

Sample Command Stream

```
%INPUT     *IN*  50
%OUTPUT    *OUT*   1
%DELETE    PIL00016
%AFTER     PIL00079
             GETSPACE  8193,T=3
             LR     15,1
             L      14,=F'8192'
%DELETE    PIL00830,2
%DELETE    PIL07044,PIL07049
             PUTLINE
             GETLINE
%AFTER     PIL09711
SAVR#        DS   6F
SAVREG#      DS  18F
PARREG#      DS   6F
%BEFORE    FILEMARK
%CLOSE
%CLOSE     *IN*
%WTM       *OUT*
%REW       *OUT*
%RUN       *IN*
%END
```

LIBRARY FILE DESCRIPTION

Name: *WATERR

Contents: WATFOR-produced error codes and their corresponding definitions. The file is approximately 370 lines. It is the same as the section WATFOR ERROR MESSAGES in the WATFOR writeup in section MTS-580.

Purpose: To provide a means of interpreting WATFOR diagnostics from a terminal.

Usage: The file may be listed using a $COPY (or $LIST) command.

Examples:
```
$COPY *WATERR
$COPY *WATERR(10,20)
```

LIBRARY FILE DESCRIPTION

Name: *WATFOR

Contents: The University of Waterloo FORTRAN IV Compiler.

Purpose: Compile and execute FORTRAN IV programs.

Usage: The compiler is invoked by a RUN command, specifying *WATFOR as the object file.

Logical I/O units referenced:

SCARDS	The source program to be compiled and its standard input unit.
SPRINT	The compiler listings and diagnostics and the source program's standard output unit.
SPUNCH	Standard punch unit. (optional)
0	Pre-stored library of WATFOR-coded subroutines. (optional)

Examples

```
$RUN *WATFOR; 5=*SOURCE* 6=*SINK* 0=*WATLIB
$RUN *WATFOR
$RUN *WATFOR; SCARDS=MYFILE 1=MYDATA
```

Description: /360 WATFOR is a one-pass load-and-go processor developed at the University of Waterloo. Because of fast compilation, and since loading time for the object program is virtually eliminated, large time savings may be realized over standard FORTRAN-G processing. WATFOR will accept FORTRAN-IV statements as described in IBM form C28-6515, subject to the extensions and restrictions noted in this writeup. The translator provides comprehensive error diagnostics at both compile time and and execution time; all violations of the FORTRAN language are reported at compile time, and the object program checks for programming errors at execution time. Most error comments will contain a 3-character code; this code may be interpreted by consulting the section entitled "WATFOR Error Messages" at the back of the WATFOR usage.

Note: The writeup on WATFOR usage will be found in section MTS-580.

LIBRARY FILE DESCRIPTION

Name:	*WATLIB
Contents:	A library of WATFOR-coded functions and subroutines.
Purpose:	To provide a library facility for the WATFOR translator.
Usage:	The file is referenced by *WATFOR through logical unit 0.
Example:	$RUN *WATFOR; 5=*SOURCE* 0=*WATLIB
Description:	The subprograms entered into WATLIB will depend on the varying needs of the WATFOR user community. Arrangements may be made through the Computing Center to enter a routine into the library. *WATLIB is formatted according to the specifications outlined in section IV of the WATFOR write-up.

LIBRARY FILE DESCRIPTION

Name: *2250EDIT

Purpose: To allow using the 2250 display console to edit a file.

Logical I/O units referenced:

SCARDS - input file

SPRINT - output file

Usage: Initially the program is in sequential mode and will display the first line in the file along with it's line number under the heading "Current Input line". Displayed also will be the next four sequential lines in the file for context. Furthermore as lines are accepted (written into the output file), they are saved and displayed under the heading "Last Five Output Lines". None of the above information may be altered by the user.

The next line displayed is the "Current Output Line". This line is initially identical to the current input line and it is the line which may be modified by using various combinations of the programmed function keyboard, alphanumeric keyboard, light pen and cursor position. An explanation of the various programmed function keys follows:

Key 16 - (accept line) - in the sequential mode writes out the current output line into the output file (using the line number of the current input line) and displays the next sequential line as the current input line (and current output line).

Key 4 - (delete line) - when used in the sequential mode, writes out a "Zero Length Line" to the output file, (using the line number of the current input line), and causes display of the next sequential line as the current input line (and current output line).

Key 9 - (insert character) "opens" a space in the current output line above the cursor position (the light pen can be used to position the cursor) to allow insertion of a character from the alphanumeric keyboard.

Key 15 - (delete character) - deletes the character above the cursor in the current output line (again the

light pen can be used to position) and "closes" the line accordingly.

Key 0 - (edit line) - "assembly edits" (columns 1, 10, 16, 30) the current output line, assuming for example that the input is in free form format.

To switch to the indexed mode of operation, the asterisk following the word "indexed" should be touched with the light pen. The program will now expect at least one "MTS line#" to be entered from the alphanumeric keyboard before the remaining options are exercised.

Key 28 - (display line) - will display as the current input line (and current output line), the line from the input file corresponding to the line number entered if such a line exists.

Key 30 - (copy lines) - in conjunction with two line numbers entered (use the jump key to enter the second line number), will copy the lines starting with the first, up to and including the last into the output file.

Key 4 - (delete line) - in the indexed mode writes out a line of length 0, using the line number entered.

Key 16 - (accept line) - in the indexed mode writes out the current output line into the output file using the line number currently displayed (i.e. It's possible to move lines by displaying, changing the line number and accepting if desired)

Inserting lines in indexed mode is accomplished by typing in a line number, using the jump key to position the cursor at column 1 of the initially blank current output line, entering the desired line and hitting the accept key.

In indexed mode, keys 0, 9, and 15 operate the same as in the sequential mode.

At any time Key 2 - (set tab) may be used to set up a "cursor tab" for the current output line. The first time the set tab key is hit a blank line will be displayed with the cursor below column 1. The user should enter the character "+" in the columns it is desired to tab to. Hitting the set tab key again registers these columns and thereafter hitting key 27 - (tab) will move the cursor to the next tab column in the current output line.

To switch back to sequential mode from indexed a starting line number should be entered and the asterisk following the word "sequential" touched with the light pen.

On the alphanumeric keyboard, "cancel" will blank the current output line ,hitting "end" twice will terminate the edit program, and "jump" will move the cursor to appropriate starting positions.

LIBRARY FILE DESCRIPTION

Name: *8ASS

Contents: Object modules of the PDP-8 assembler

Purpose: To assemble PDP-8 programs

Example Usage:

```
$RUN *8ASS; 1=*SOURCE* 2=*8OPS 6=-F
8=*SINK* SPUNCH=*PUNCH*@BIN
```

Description: The two-pass assembler reads a set of standard definitions from "2". EBCDIC source lines are read from "1" and Pass I intermediate storage is listed on "6" (a 93-character record for each input line). Pass II input is from "6"; the assembly is listed on "8" and binary cards are produced on "SPUNCH".

The input format of the assembler in 8SS (see 7090 MESS Manual, volume 2) is compatible with 8ASS format.
The internal symbol table size of the assembler is fixed. If a program defines S symbols and references symbols (including standard opcodes and procedure calls) R times, then the PDP-8 assembly size is limited by:

$$10(S+65)+2R<M$$

For *8ASS, M = 30,000

8ASS ASSEMBLY LANGUAGE FORMAT

Eighty-character source lines are scanned, left to right, for three fields: label, opcode, and operand. With the following exceptions, fields are delimited by blanks.

The label field may contain an alphanumeric string less than 9 characters long, starting with a letter as the first character of the line. If the first character is a blank, the label field is empty.

The operand field must begin before character 17; otherwise it is empty.

During the scan, occurrence of a semicolon (";") will force the remaining fields to be empty.

After all three fields are scanned, the rest of the line is treated as comment.

Note: For further information, see the 8ASS writeup in section MTS-590.

LIBRARY FILE DESCRIPTION

Name: *8SSPAL

Contents: object module to convert PDP-8 load card images to PDP-8 paper tape formats.

Usage: *8SSPAL is used to convert the load cards produced by the 8SS system on the 7090 or the output of the 360 PDP-8 assembler (*8ASS) to the paper tape formats required by most PDP-8 configurations.

Logical I/O units referenced:

SCARDS	Source of binary load cards
SPUNCH	Sink for paper tape output

Examples:
```
$RUN *8SSPAL; SCARDS=CARDSOURCE SPUNCH=TAPESINK
$RUN *8SSPAL; SPUNCH=*PAPERTAPEPUNCH*@BIN
```

Note: Both the input and output FDnames refer to entities which must have the binary attribute associated with them. Thus if CARDSOURCE is an intermediate file, $COPY *SOURCE*@BIN TO CARDSOURCE@BIN would load it properly. Similarly $COPY TAPESINK@BIN TO *PAPERTAPEPUNCH*@BIN would punch the output. *8SSPAL both reads and punches @BIN.

THE DYNAMIC LOADER

The dynamic loader takes the output from the language processors such as the Assembler or FORTRAN compiler and puts it into the memory of the computer so that it can be executed. The loader is invoked by the **$RUN** and $LOAD commands and by the system subroutines **LINK**, **LOAD**, and **XCTL**. Although the general processing done by the loader is the same in all cases, some details differ depending upon the manner in which the loader is invoked.

The writeup for the dynamic loader consists of several sections, each covering a different aspect of the loader. A list of these sections follows:

SECTION NUMBER	TITLE
MTS-292	Description of the Loading Process
MTS-294	Description of the Loader Input
MTS-296	Description of the Loader Output
MTS-298	Loader Library Facility
MTS-341	Loader Internal Specification

DESCRIPTION OF THE LOADING PROCESS

Introduction

The dynamic loader takes one or more object modules, each consisting of one or more control sections, and puts them into the memory of the computer. External symbols which are referenced by the object modules but not defined by any of them are looked for in the system library, and loaded if found. The program thus loaded can be executed, provided that no fatal errors were detected during the loading process.

Loader Input

The dynamic loader receives its input from two sources. The first source is the file or device specified as the loader input parameter on the $RUN or $LOAD command or the call to LINK, LOAD, or XCTL. The information contained in the loader input file or device should be loader input lines as described in MTS-294. Each line is read and processed sequentially until an end-of-file, a load terminate line, or a line which is not a legal loader input line is encountered. At this point the loader will no longer reference input from this source.

If after processing the input from the user specified source, external symbols remain which have not yet been defined although they are referenced, the loader continues with its input coming from the system library, which is contained in the file *LIBRARY. The object modules from the library are loaded selectively. That is, only object modules which satisfy unresolved external symbols and modules which are in turn referenced by such modules are loaded.

Optional libraries may be concatenated to the end of the user specified source. Object modules from the optional library will be selectively loaded after all modules preceding it in the user specified source have been loaded and before the system library is scanned. The optional library may be either a public library, as the Scientific Subroutine Package in the file *SSP, or a private library created by the user. More details about optional libraries can be found in section MTS-298.

Resident System Symbols

After the object modules from the user specified source and the library have been loaded, there may still be some unresolved symbols which

reference the resident system. Examples of such symbols are GETSPACE, READ, SPRINT and ERROR. The correct values for these symbols are located in two system tables, one containing definitions of symbols whose values never change, the other symbols which change from user to user such as SPRINT. These tables are searched in an attempt to resolve all symbols not yet defined. Symbols which are still undefined after this search are printed out in error messages as undefined and loading halts.

If the loader is invoked via a call to one of the subroutines LINK, LOAD, or XCTL, symbols and their values may be added to the system table of unchanging symbols (the initial ESD table) and entirely new tables of changeable symbols (low core symbol dictionaries) may be created by the user. For details on this, see the appropriate subroutine writeups, the description of the LCS loader card in MTS-294, and the loader library specifications in MTS-298.

Loader Output

The dynamic loader generates two types of output as the fruits of its labors. The first type of output is the program which remains in the memory of the computer. This program is a representation of the information in the object modules which have been loaded. The representation is such that the program can be executed, provided that no fatal errors were detected during the loading process.

The second type of output generated by the loader consists of printed information describing the results of the loading endeavor. This includes error messages, a map showing the value of each external symbol as well as the location and relocation factor of each control section, and the entry point of the program. The map may be omitted.

Entry Point Determination

The entry point of a program is the point within the program where execution will begin if the program is executed. That is, it is the location of the first instruction within the program to be executed. This location is determined by the loader as it processes the object modules forming the program. The following algorithm is used to determine the entry point:

1. If one or more entry lines (ENT) have been encountered, the entry point is the value of the external symbol referenced on the first such ENT line.

2. Otherwise; if the processing of input from the user specified source was terminated by a load terminate line (LDT) and that line referenced an external symbol, the entry point is the value of that external symbol.

3. Otherwise; if one or more object module end lines (END) contain an entry point address, the entry point is the value of the entry point address from the first such END line.

4. Otherwise; if at least one control section has been encountered, the entry point is the address of the first byte of the first control section encountered.

5. Otherwise; the entry point is zero.

Loader Processing Details

As the loader processes its input, it maintains its own internal tables and performs other tasks necessary for the generation of an executable program in memory. Following is a list of the various tasks performed by the loader.

1. The loader maintains an external symbol table which contains an entry for each external symbol thus far referenced or defined. Since there may be many private control sections, there may be many entries in the table whose names are blank. The information stored in the table for each external symbol includes the symbol name, the location of the symbol if it has been defined, the relocation factor of the symbol if it is the name of a control section, the current "short hand name" or external symbol identification (ESID) by which the symbol is referenced in the lines of the object module currently being loaded, and a set of status bits indicating if the symbol is defined, if it is the name of a control section, common section or pseudo-register, if the symbol was defined by the system, and so forth.

2. Address constants which refer to external symbols are relocated. If the symbol has not yet been defined when the address constant is encountered, pertinent information is stored in a relocation table so that the relocation can be performed when the symbol is defined.

3. Storage is acquired in the memory of the computer for each control section encountered in the loader input, and the text for each of these control sections is loaded into the storage acquired for the corresponding control section.

4. An entry point is determined via the algorithm described above.

5. The loader input is checked for errors. If errors are detected an error message is printed and loading is either continued or terminated depending upon the severity of the error.

6. A map is produced showing pertinent information about each external symbol and control section name. The map may be omitted as an option of the user.

7. The loader is capable of producing a reference table for the external symbols, showing the locations at which each external symbol was referenced. However, there is currently no way in MTS to turn this option on.

Loader Invocation Details

The loader is invoked by the $RUN and $LOAD commands and by the system subroutines LINK, LOAD, and XCTL. Although the general processing done by the loader is as described above in all cases, some details differ depending upon the manner in which the loader is invoked. This section presents a general outline of the processing performed by each type of invocation of the loader. For more details, see the appropriate command or subroutine writeup.

A. Invocation by a $RUN command.

1. Storage and devices acquired by previous commands and all their consequences are released.
2. Devices required for the run are acquired.
3. The program is loaded, with the map produced if a loader output file or device was specified.
4. The registers are set up as in a normal subroutine call.
5. The system transfers to the program at the entry point.
6. The program can return to the system via normal subroutine return conventions or by calling the system subroutines SYSTEM or ERROR.

B. Invocation by a $LOAD command.

This is processed exactly as a $RUN command, except that the loaded program is not executed. Instead MTS remains in command mode. The program will be started in the normal fashion if a $START command is given. This allows the user to display and alter sections of the loaded program before execution begins.

C. Invocation by a call upon LINK.

1. The program is loaded, with the map produced if a loader output file or device was specified on the initial $RUN or $LOAD command.
2. The registers are set up to make the call upon the LINK subroutine transparent.

3. The system transfers to the program at the entry point.

4. If the program returns as a normal subroutine, all storage acquired to load it plus all further storage acquired by it at its storage level and at higher levels is released.

5. The system returns to the program which called LINK.

D. Invocation by a call upon LOAD.

1. The program is loaded, with the map produced if a loader output file or device was specified on the initial $RUN or $LOAD command.

2. The external symbol table and entry point of the loaded program are made available to the program which called LOAD.

3. The system returns to the program which called LOAD.

Note that the program loaded is in core and ready to use. All storage acquired during the loading operation is cataloged under a unique storage index number. This storage can later be released by calling the subroutine UNLOAD.

E. Invocation by a call upon XCTL.

1. The storage having the current storage index number is released. This presumably is the program which is calling XCTL.

2. The program is loaded, with the map produced if a loader output file or device was specified on the initial $RUN or $LOAD command.

3. The registers are set up to make the call upon the XCTL subroutine transparent.

4. The system transfers to the program at the entry point.

5. If the program returns as a normal subroutine, the action taken is the same as what would have happened if the program which called XCTL had returned instead.

DESCRIPTION OF THE LOADER INPUT

The loader accepts as valid input thirteen different types of input records. These input records break into three groups: one group of records is generated by language translators to form object modules, the second group is generated by the user to "patch" object modules and specify entry points, while the third type is used to control selective loading of library object modules and other services.

All loader input images are identified and referred to by the sequence of three characters appearing in the second, third, and fourth columns of the input record. Although loader input usually consists of card images and this writeup shall describe them as such, variable length input lines are accepted by the loader provided that all required information is present as specified and all counts are correct. Records of up to 256 bytes in length are accepted.

Following is a brief paragraph describing each of the valid loader input records. If the loader encounters a non-loader type record, it prints out the contents of the record and considers it to be a logical end-of-file. Following that is a table showing the exact record formats for each type of loader input record.

1. Translator-generated Load Records

A. ESD Input Record. (External Symbol Dictionary).
Translators generate ESD records in object modules to define the external symbols contained within the module and indicate which symbols external to the module are referenced within it. All necessary information, such as length of control sections and program common, alignment of pseudo-registers and address of an external symbol definition within a control section is specified for the loader.

B. TXT Input Record (Text)
The TXT input records generated by translators contain the actual text or data of the module. Information on the TXT record includes the control section the data is contained in, the starting address of that data within the control section, the number of bytes of data on the record and the record itself.

C. RLD Input Record (Relocation Dictionary)
The RLD input records generated by translators contain all information necessary to relocate address constants which refer

to external symbols or labels within the control section. Information on the RLD record includes the control section and address within it of the address constant, the length of the address constant, and the external symbol which the address constant referred to.

D. END Input Record
The END record indicates the end of the object module. It may optionally give an entry point address within the module.

E. SYM Input Record
The assembler generator SYM records in object modules if the TEST parameter is turned on. These records contain a dictionary of symbols which are internal to the module and are used by symbolic debugging packages. These records contain no information that is vital to the loading process and are ignored by the loader.

2. User-generated Load Records

A. LDT Input Record (Load Terminate Record)
A load terminate record is optional, but may be placed after the last object module by the user. The loader will never read past the load terminate record, so that it can be used to stop the loader when data records follow the object deck. It also can be used to specify the entry point of the program. The external symbol which indicates the entry point is ignored if it is blank.

B. REP Input Record (Replace Record)
A REP record can be used to "patch" (correct) errors in the text of a control section. All fields of a REP record are specified as hexadecimal digits, so that the information can be typed at terminals and is readable. The record contains the ESD identifier of the control section and starting address within it to be modified. Furthermore, it contains the data which is to replace the original text. This data must always specify an integral number of bytes of data, with two hexadecimal digits per byte. The data can be delimited by commas on the byte boundaries, if desired; and the data field ends with the first blank. The "patch" can be applied on any byte boundary and be of any length which will fit on a loader input record.

C. DEF Input Record (Define External Symbol).
A DEF record can be used to define an external symbol within an object module. Its primary use is to define symbols as external when this was forgotten in the original source module. The

record contains the name of the symbol, the control section it is located in, and its address within the control section. All numbers are specified as hexadecimal digits.

D. ENT Input Record (Entry Point Record)
An entry point record may be used to specify the entry point of the program. The record contains the external symbol with which execution of the program is to begin. ENT records are generated by some translators.

E. NCA Input Record (No Care Record)
A no care input record can be used to specify that the user does not care if the symbol referenced on the NCA record is undefined. The loader will not print an error message if this symbol is referenced but not defined. Instead, it will be treated as if it has a value of zero. NCA records are generated by some translators.

3. Library Control Records

A. LCS Input Record (Low Core Symbol Table).
The LCS record is used to present a low core symbol dictionary to the loader. The external symbol specified is assumed to be the origin of an external symbol dictionary, which contains entries for symbols and their values. If any symbol in the table has been referenced but not yet defined by the object modules loaded thus far, it is given the definition contained in the table. See section MTS-298 for a description of an external symbol dictionary.

B. LIB Input Record (Library record)
The LIB record is used to control the selective loading of library object modules. If the symbol on the LIB record has been referenced by object modules loaded thus far but never defined, the object module immediately following the LIB record is loaded. Otherwise it is ignored. Optionally the LIB record may contain the line number on which the object module begins. The object module then would not be located after the LIB card but would be pointed by it. The processing of the library is faster if this option is taken, since object modules which are not referenced need not be read.

C. RIP Input Record (Reference If Present Record)
The RIP record is used to solve problems of multiple entry points and forward references within the one-pass library scan. If the

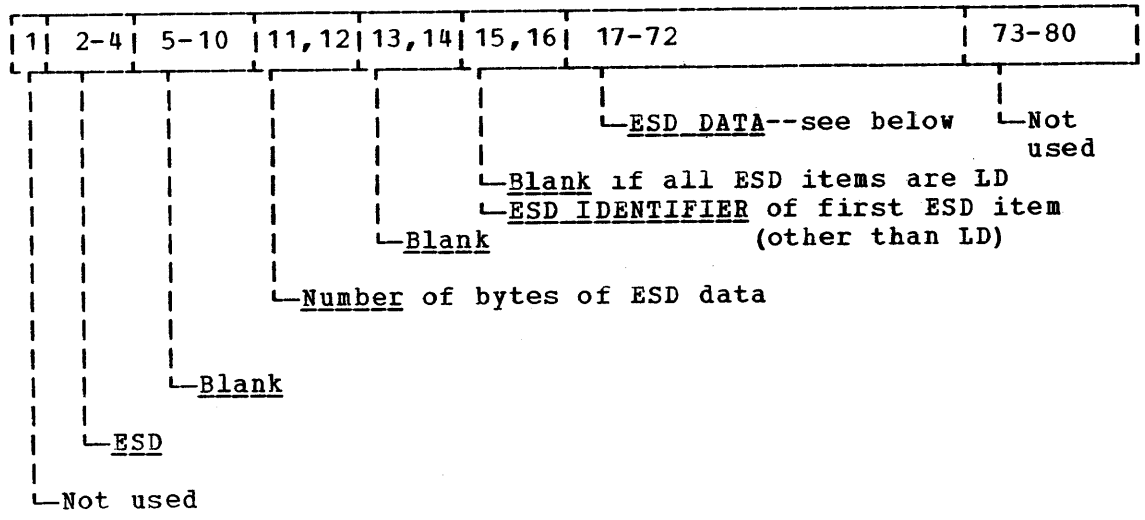
second symbol on the record has been referenced but never defined by the object modules loaded thus far, the first symbol is also marked as having been referenced but not yet defined (provided it is indeed not yet defined).

Record Formats - Dynamic Loader

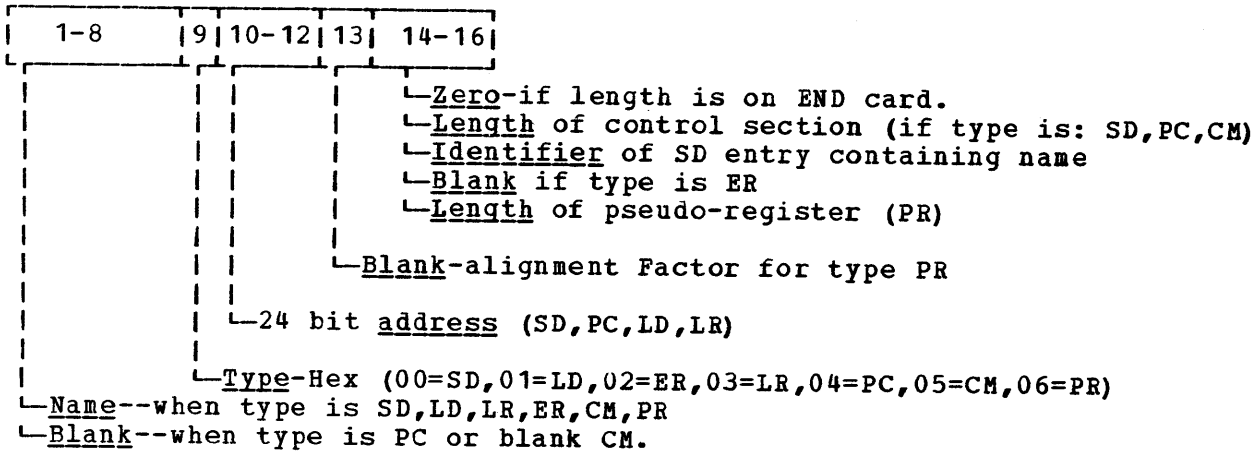
The following are the card image formats for the dynamic loader. The loader will accept variable length input up to 256 bytes long, but card images are the most common and hence are shown here.

1. Translator-generated Load Cards:

ESD Input Record (Card Image)

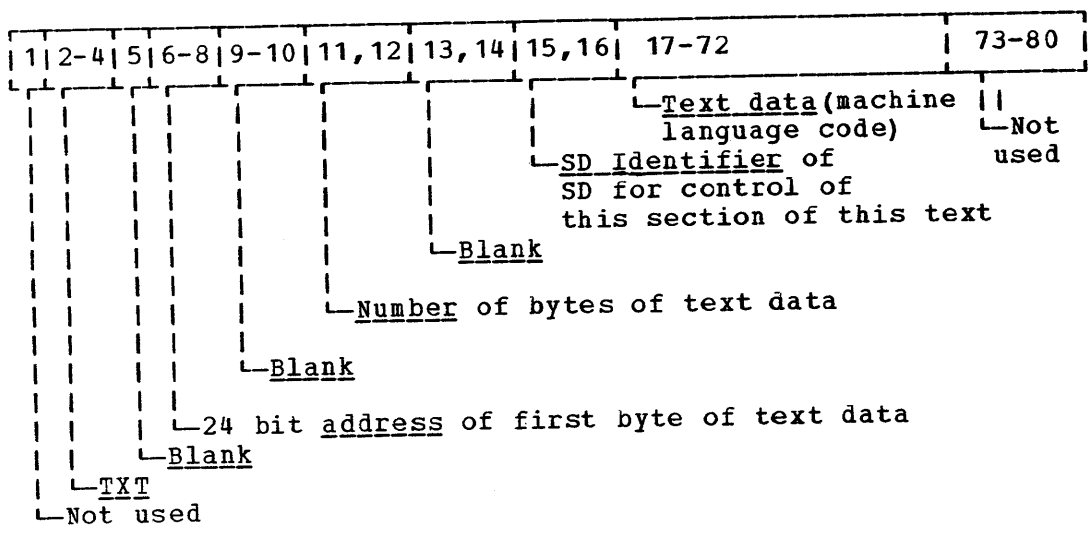


ESD Data Item

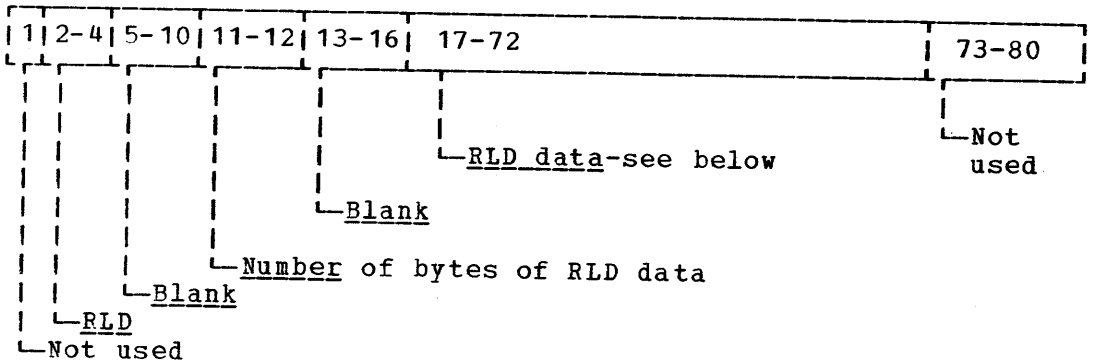


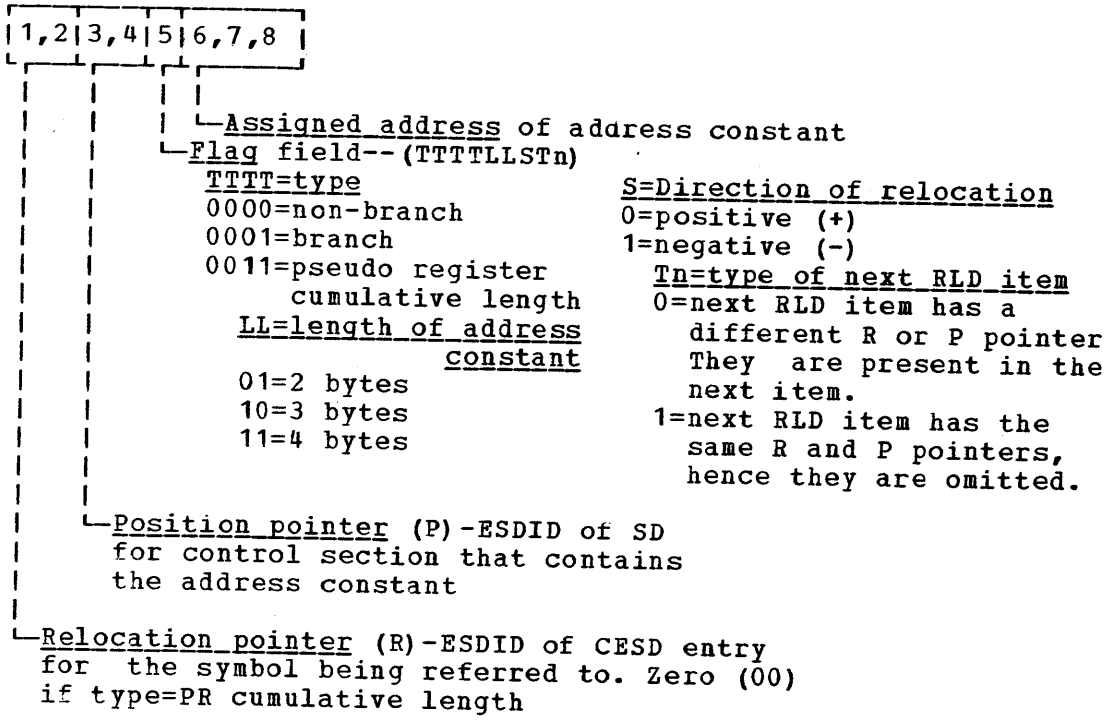
TYPE MNEMONIC	MEANING
SD	Name Control Section
LD	Entry Point Name
ER	External Symbol Name
LR	Label Reference
PC	Private(blank-name)Control Section
CM	Program Common Name
PR	Pseudo-registers Name

Text Input Record (Card Image)

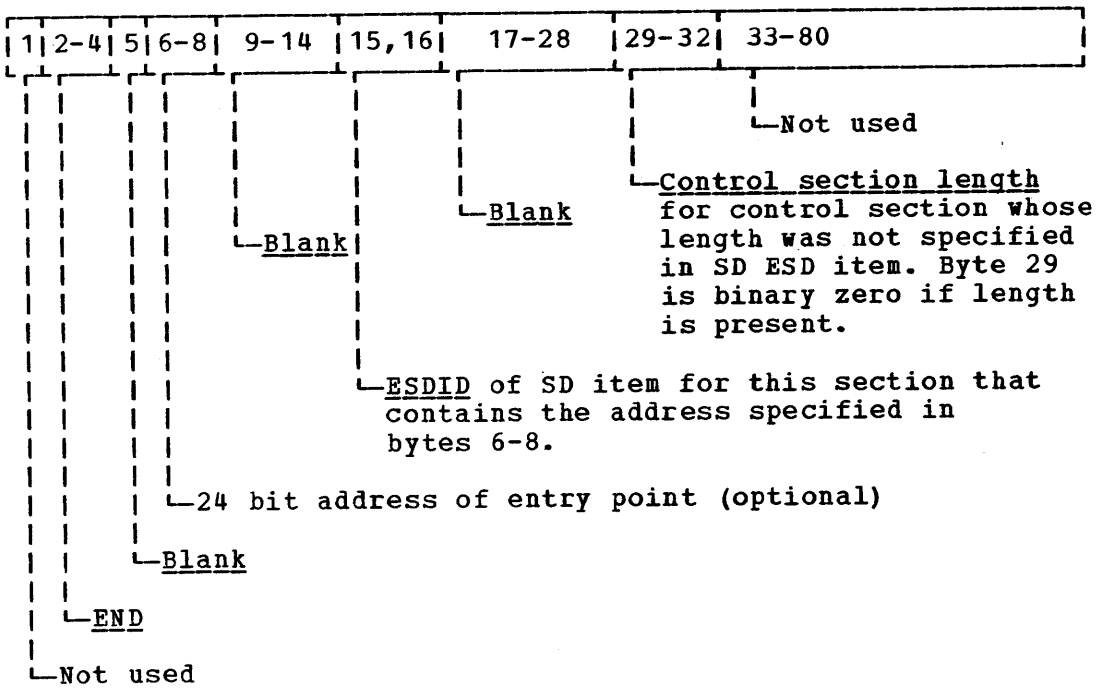


RLD Input Record (Card Image)

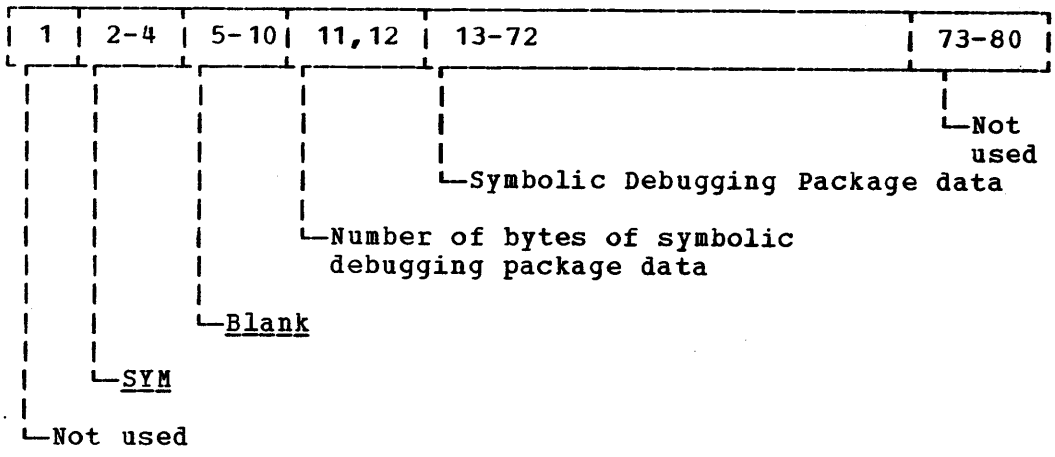




END Input Record (Card Image)

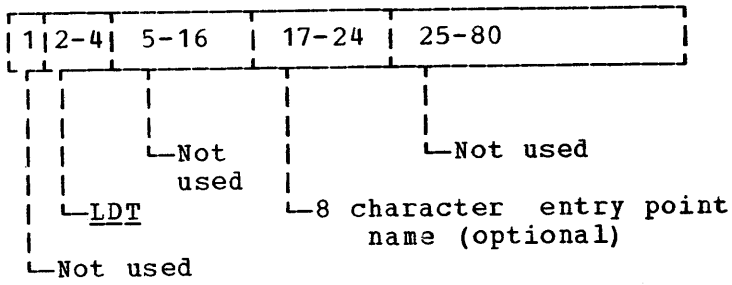


SYM Input Record (Card Image)

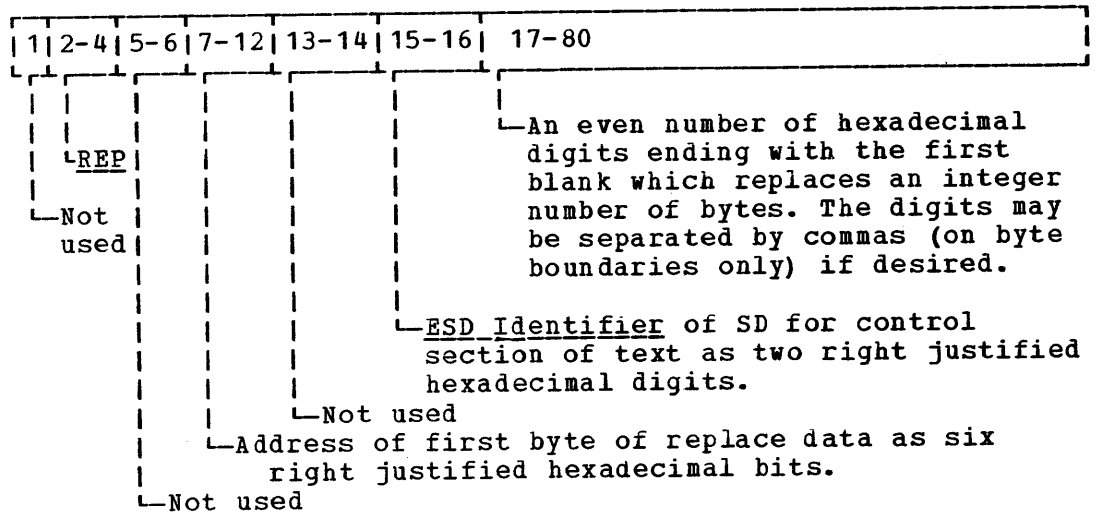


2. User - generated Load Cards:

LDT Input Record (Card Image)

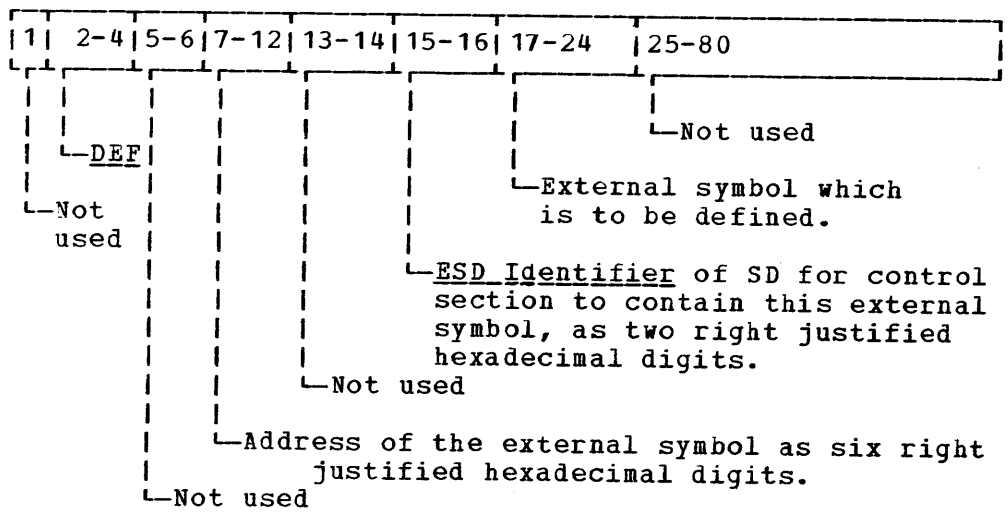


REP Input Record (Card Image)

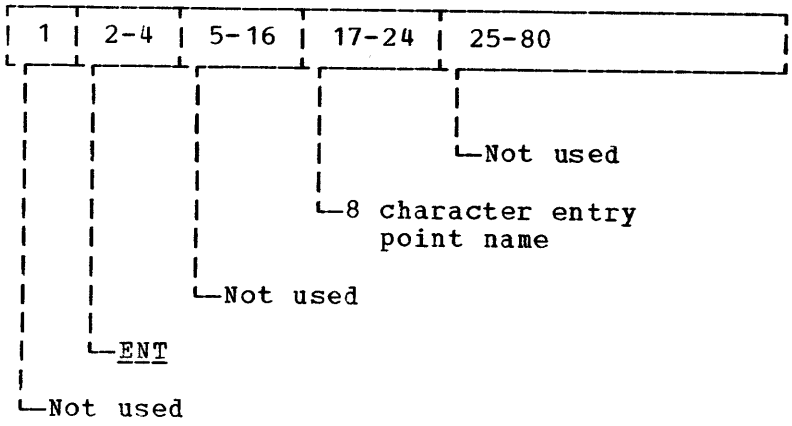


Note: A zero ESD Identifier always indicates that the address is absolute, that is, not to be relocated at all.

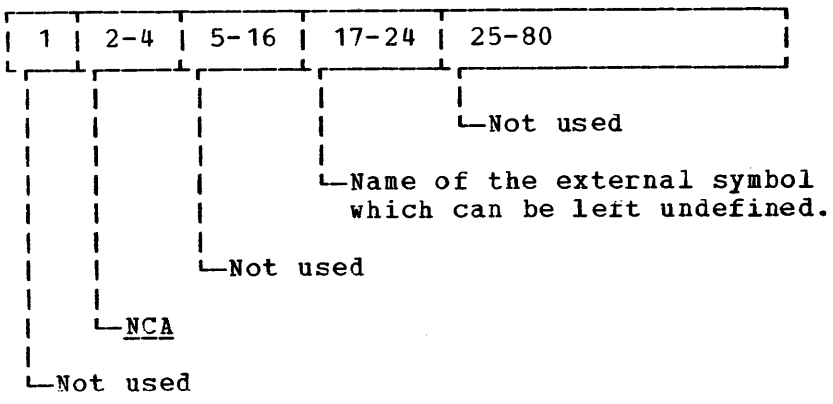
DEF Input Record (Card Image)



ENT Input Record (Card image)

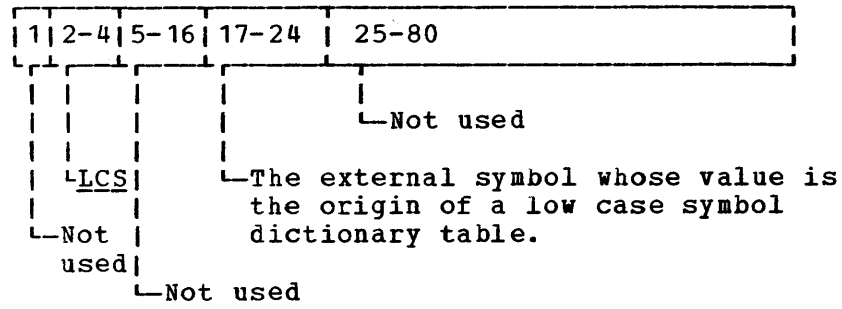


NCA Input Record (Card Image)



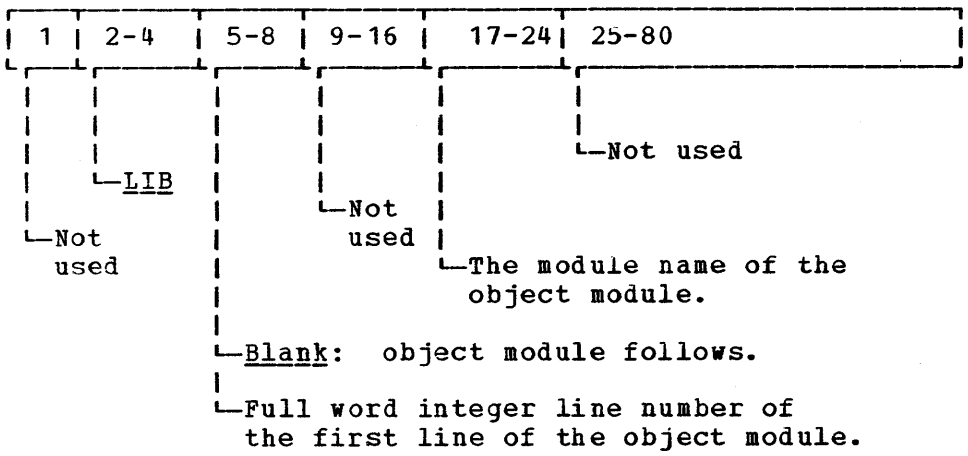
3. Library Control Cards

LCS Input Record (Card Image)

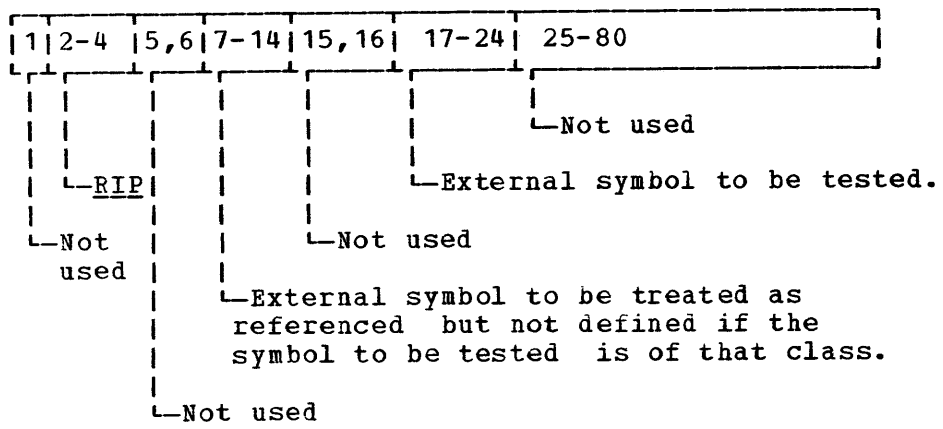


Note that the system low core symbol dictionary has the pre-defined name LCSYMBOL. An LCS card referencing that symbol can be placed at the end of the deck by the user and will eliminate several seconds of loading time if no library subroutines are referenced.

LIB Input Record (Card Image)

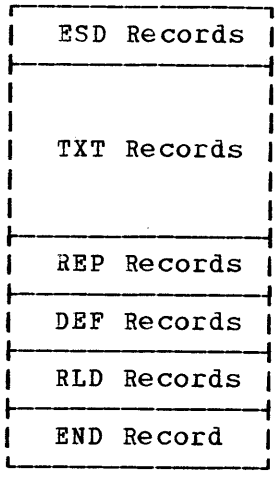


RIP Input Record (Card Image)



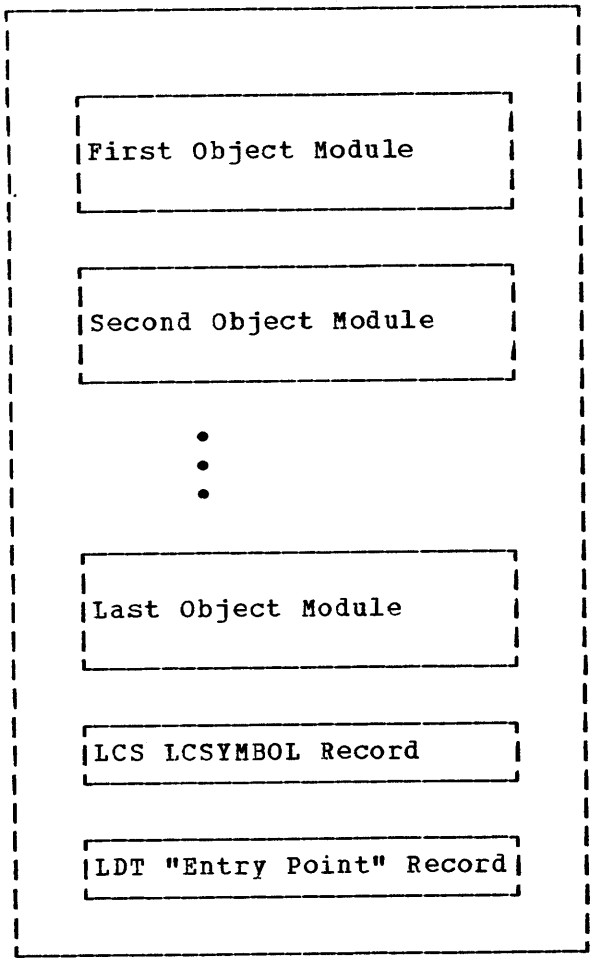
Loader Input Deck Ordering and Restrictions

The usual order of loader lines within an object module is depicted in the following diagram. The top of the diagram is the front of the object module. REP, DEF, and RLD records are optional and appear only if required. The user must supply REP and DEF cards if he requires them. Language processors do generate the object module cards in the correct order.



Order of loader lines within an object module.

The usual order of the object modules and other loader lines within the loader input is depicted in the following diagram. Because they are usually used only by the library, the positioning of LIB and RIP cards is not shown here.



Order of loader lines within the input to the loader.

A user is posed with the problem of presenting the loader will all of the loader input records in the correct order. One solution of this is to concatenate all of the files which contain object modules that are to be loaded together. For instance, suppose that three object modules have been generated by language translators, one object module in each of the files -OBJ1, -OBJ2, -OBJ3. Then an appropriate run command to load and run the contents of these files might be:

```
$RUN   -OBJ1+-OBJ2+-OBJ3
```

The contents of these files could be combined into one file if desired, by a command as:

```
$COPY   -OBJ1+-OBJ2+-OBJ3 TO -OBJECT
```

The load specified above could then be written as:

```
$RUN   -OBJECT
```

The object modules could also be put into the same file by use of appropriate starting line numbers on the output device for the translator. For instance, the object modules from two runs of the assembler could be run using the following sequence:

```
$RUN  *ASMBLR; SPUNCH=-OBJECT  SCARDS=...
$RUN  *ASMBLR; SPUNCH=-OBJECT(LAST+1) SCARDS=...
$RUN  -OBJECT
```

These hints have indicated just a few of the more common ways a user may present the loader with the appropriate set of input records.

A deck which follows all the suggestions above concerning deck ordering may still run into difficulties, especially if it uses program common or pseudo registers. There are also some problems with the ordering of CSECT, START, ENTRY, and V-type adcons within assembler language source modules. A more precise list of the deck ordering restrictions follows, with a brief description clarifying the restriction if necessary. In general, the restrictions are those which would be normally expected from a one-pass loader with special provisions for external symbol forward references.

1. Object Modules cannot be intermixed.

   The moral is, do not shuffle your object module decks together. Object modules should be placed in sequential order, one after the other.

2. If more than one object module references a program common or pseudo-register, the object module which declares the program common or pseudo-register to be the largest should appear first.

   The first reference to a program common or pseudo-register is used by the loader to assign the space for that program common or pseudo-register. Hence if a later reference requires more space, it will run over the end of the space actually allocated. No error messages are generated for this situation.

3. An LCS record may appear anywhere; however, it will define only those symbols which are referenced in the object modules which preceded it.

4. An LDT record must, of course, be the last record, since it "turns off" the loader.

5. Within an object module, an ESD Identifier must be defined before it is referenced.

   The ESD Identifiers are integers which are put into correspondence with the external symbols referenced within the object module, so that all references to these symbols on the other records within the module can consist of a two byte integer rather than the eight

byte symbol name. On the surface this restriction merely states that TXT, REF, DEF, and RLD records should not appear before the ESD records. More subtly though, some items on the ESD records (namely items generated by the ENTRY statement in assembly language) refer to another ESD item (namely the control section containing the definition) via an ESD identifier. Hence a sequence in which the ENTRY statement appears before the START or CSECT statement for the control section containing the symbol will generate an ESD Identifier forward reference within the ESD records themselves. That is, the sequence

```
        ENTRY  WHEE
        CSECT
WHEE    EQU    *
```

will cause a loader error, whereas the sequence

```
        CSECT
        ENTRY   WHEE
WHEE    EQU    *
```

will be loaded correctly. This problem in forward references of ESD Identifiers should not be confused with external symbol forward references, which works correctly in all cases.

6. Within an object module, no external symbol may be assigned more than one ESD Identifier.

   An external symbol will be assigned more than one ESD Identifier if it occurs more than once on the ESD records. The only known way of causing this error is to refer to the same symbol in a V-type adcon and in a CSECT, START, EXTRN, or ENTRY statement in assembly language. That is, the sequence

```
WHEE    CSECT
        •
        •
        •
        DC    V(WHEE)
```

Will cause a loader error, whereas the sequence

```
WHEE    CSECT
        •
        •
        •
        DC    A(WHEE)
```

will be loaded correctly. The reason is that the former construction declares the external symbol WHEE to be defined in this object module and external to this object module at the same time, which confuses the loader.

7. The TXT and REP records within an object module take effect in the

order they appear. An RLD item either takes effect immediately if the external symbol referenced is already defined, or is saved until the object module which defines that external symbol has been processed.

8. The last record of an object module must be its END record.

DESCRIPTION OF THE LOADER OUTPUT

Introduction

The dynamic loader produces two types of output as the fruits of its labors. The first type of output is the program which remains in the memory of the computer. The second type of output consists of printed information describing the results of the loading endeavor. Both of these are described in more detail below.

The Program

The sole purpose for the existence of the loader is to put programs into the memory of the computer in such a way that the program can be executed. The loader performs three operations to each control section that is loaded. These are:

1. The loader acquires storage for each control section. An independent block of storage is acquired for each control section. The block of storage acquired for a control section comes from any available free area of the computer's memory which is large enough. Hence it is unlikely that any two control sections will be loaded in contiguous storage or that the loading of the same program on two different occasions will produce an identical map. This is a very important difference between MTS and other systems which have been run at the University of Michigan in the past. It is not at all possible to postulate where a control section will be loaded relative to other control sections. Core storage is acquired dynamically on a competitive basis against the other users of the system.

2. The loader inserts the text of each control section into the area acquired for the corresponding control section. Of course, the relative displacement between any two bytes of a control section is preserved. One can think of this process as picking up the text of the control section and copying it into the storage acquired for the control section byte-by-byte, in the exact order of the original control section. No changes are made at this point to the text of the control section to indicate that is has moved.

3. Because control sections can be located anywhere arbitrarily relative to each other, it is necessary for the loader to be able to "inform" a control section of the location of itself and other control sections which it wishes to reference. Because of the

base-displacement addressing scheme on the IBM/360, only address constants and channel command words need to be modified by the loader. This process is referred to as relocating the appropriate address constants and channel command words. Address constants and channel command words which require relocation contain expressions which have a machine address as a value. Furthermore, these expressions contain references either to external symbols or symbols within the control section containing the address constant or channel command word. The loader evaluates these expressions, using as the values of these symbols the memory locations they have been assigned to in the loading process. The value of this expression then becomes the contents of the address constant or channel command word. In this manner then, a control section may refer to other control sections even through they are assigned storage independently.

Printed Output

The second type of output generated by the loader consists of printed information describing the results of the loading endeavor. This includes error messages, a map showing the value of each external symbol as well as the location and relocation factor of each control section, and the entry point of the program. All numbers printed in the map and error messages are in hexadecimal. The map may be omitted. A sample of some loader output is shown below. The various parts of the printed output will be discussed following the example.

Sample Loader Printed Output

```
... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...

SNARK     IS AN UNDEFINED SYMBOL.
ENTRY = 047018

NAME      VALUE  T RF     NAME      VALUE  T RF     NAME      VALUE  T RF

SNARK     ------          SYSTEM    014CAC *        ERROR     014CE4 *
GDINFO    017810 *        GETFD     0181E0 *        SDUMP     0183D8 *
SCARDS    018464 *        SPRINT    018476 *        SPUNCH    018488 *
SERCOM    01849A *        READ      018506 *        WRITE     018522 *
LCSYMBOL  018BA8 *        MYCOM     02C018 C02C018  SUB1      02C730
SUB1#     02C730   02C730 IHCFCOMH  030018  030018  IBCOM#    030018
FDIOCS#   0300D4          MAIN      047018          MAIN#     047018  047018

... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
```

The Entry Point

The entry point of the program is computed by the loader as described in the section MTS-292. This entry point is printed as a part of the loader output. In the sample above the entry point is at address 047018, which is also the value of the external symbol MAIN. Hence execution of the program will begin in the subroutine whose name in MAIN. This effect might have been caused by an LDT record giving the entry point external symbol "MAIN".

The Map

The map contains an entry for each external symbol which has been referenced in the loading process. Each entry gives the name, value, type, and relocation factor of its symbol. The entries of the map are printed in ascending order by value. The information in an entry is described in more detail below:

1. Name
   This is the eight character name of the external symbol. Each non-blank name can be associated with only one external symbol.

2. Value
   This is the actual address assigned to the external symbol. It is the address in the computer's memory where the instruction or data associated with the external symbol is located. If the symbol is undefined the value is printed as six dashes. For pseudo-registers this is the displacement assigned within the pseudo-register area.

3. Type
   The type gives some indication of the status or special significance of the symbol. The allowable types and their meanings are:

   * The symbol is a system symbol. It is either pre-defined by the system, or received its value via the processing of an LCS record.

   C The symbol is the name of a program common. Note that program common is treated much like a control section. There is no fixed area in which program common is located and there may be more than one common.

   D The symbol has been defined by the user with a DEF control record.

   P The symbol is a reference to pseudo-register storage. This is used by some translators to handle dynamic storage assignments.

   All other symbols are given a blank type code. This indicates

that they are the ordinary type of symbol with no special meaning. For instance, control section names and entry point names fall in this category.

4. Relocation Factor

Every symbol which is the name of a control section also has a relocation factor printed in its entry. The relocation factor is the number which must be added to addresses in an object or assembler listing to compute the actual machine memory location address of the item in question. That is, the relocation factor is the location corresponding to the location 000000 in the control section object listing if indeed there was such a location. The relocation factor is essential for finding information in storage dumps.

Error Messages

As the loader is processing its input, it checks the input for errors. An error message is printed for each error detected and the action taken next depends upon the severity of the error. If the error is a non-recoverable error, loading is terminated immediately and execution of the program is aborted. If the error is is a fatal error, loading is continued but the execution of the program is still aborted. Finally, if the error is a non-fatal error, loading continues as normal. A list showing each message, its severity, and likely causes for the message is given below:

1. INPUT RECORD LONGER THAN 256 CHARS.

   Non-recoverable. An input record was read which was longer than 256 characters. Since the input buffer is only 256 characters long, internal tables may have been destroyed.

2. CARD NOT A LOAD CARD, END-OF-FILE ASSUMED.

   Non-fatal. An input record which did not have a legal loader record type in positions 2,3 and 4 was read. The contents of the record are printed, and are lost for further processing. The loader continues as if this had been and end-of-file.

3. PROGRAM BUFFER OVERFLOW.

   Fatal. The core storage acquired for the loading of the next control section is not large enough to hold the control section. This error is intercepted by MTS and treated as a request for more space by the loader. It should never appear.

4. XXXXXXXX IS MULTIPLY DEFINED. FIRST DEFINITION USED.

   Non-fatal. Two object modules containing the definition of the given external symbol have been loaded. The first definition of the

symbol is the one actually used in all references to the symbol. The second definition is ignored entirely in loading.

5. ESD/RLD TABLE AREA OVERFLOW.

Non-recoverable. The area used by the loader for internal tables is not large enough to hold all the information necessary to load your program. You may take one of the following actions to reduce the table requirements of the program:

a. Re-order your program so that frequently referenced external symbols are defined before they are referenced. This will reduce the amount of space required to hold RLD information.

b. Reduce the number of external symbols used by your program.

c. Pre-process your program through a LINKAGE EDITOR to reduce both the number of control sections and amount of unresolved RLD information.

6. ESID XXXX RELOCATION FACTOR UNDEFINED.

Non-recoverable. Information on an ESD, TXT, REP, DEF, RLD, or END record is to be relocated relative to the symbol which has the specified ESID. However, either no symbol has that ESID, or the symbol which does have that ESID is not the name of a control section and hence does not have a relocation factor associated with it. Common causes of this error are

a. Mixing up the order of the object module records.

b. Leaving object module records in a file which contains a newer, but shorter object module so that the loader attempts to process these old, obsolete records.

c. Placing the ENTRY pseudo-op before the CSECT pseudo-op for the CSECT containing the external symbol in assembly code.

D. Referencing an external symbol which is defined within the current assembly with a V-type adcon.

7. ESID XXXX UNDEFINED.

Non-recoverable. An address constant or channel command word references the symbol with the specified ESID within it's expression. However, no external symbol has been given this ESID in the object module being processed. This message is caused by the same errors as listed under error message 6 above.

8. XXXXXXXX IS AN UNDEFINED SYMBOL.

Fatal. The external symbol specified has been referenced by at least one of the object modules loaded, but has not been defined.

MTS Errors or Program Interrupts During Loading

The loader does not check the address specified on TXT and REP records to see if the address is actually within the legal range for the control section. Instead, it dutifully loads the data on the record as requested. An illegal address may cause the loader to generate a program interrupt. The cause of the interrupt will usually be an addressing or protection violation. The address of the instruction at fault will be within the first few pages of the machine memory.

It also is possible that MTS errors will develop during loading which usually are not the fault of the loader. For instance, a bad file will cause an error.

LOADER LIBRARY FACILITY

The dynamic loader has associated with it a very primitive library facility. The facility consists of three control records, namely LCS, LIB, and RIP, records. The LCS record is used to cause referenced but not yet defined symbols to be defined from an in-core table if they are defined there. The LIB record is used to selectively load the object module which follows or is pointed to by the LIB record only if the module name has been referenced but not yet defined. The RIP record is used to handle forward references and multiple entry point problems in the one-pass library. A set of symbols may be pre-defined. These are defined at the initialization of the loading process and cannot be re-defined by the user.

The System (Public) Library

The MTS system library facility consists of an in-core dictionary of resident symbols located at LCSYMBOL and a file of object modules with the appropriate control records in the file *LIBRARY. The action taken by the loader with respect to system symbols is as follows:

1. A small set of pre-defined resident system symbols are defined. These cannot be redefined by the user.

2. The user's object modules are loaded.

3. If after loading all of the user's object modules any symbols remain undefined:

    a. A pass is made of the file *LIBRARY to selectively load object modules which define thus far undefined symbols.

    b. The table at LCSYMBOL is searched for any symbols still undefined.

Note that if only pre-defined and resident system symbols are referenced, the pass over the library can be eliminated by enclosing an LCS record referring to LCSYMBOL after the last object module. Several seconds can be eliminated from the loading time in this manner.

Optional Libraries

Optional libraries may be concatenated to the end of the user specified source. Object modules from the optional library will be selectively loaded after all modules preceding it in the user specified source have been loaded and before the system library is scanned. The optional library may be either a public library, as the Scientific Subroutine Package in the file *SSP, or a private library created by the user.

A private optional library consists of the object modules the user desires in his library together with the necessary LIB and RIP control records to define the module names, entry points, and references for the selective loading feature of the loader. Although the user can construct such a library himself by inserting appropriate LIB and RIP records in with his object modules, this task has proven formidable enough with large libraries that a program has been written that analyzes the object modules that form the library and generates the library complete with all LIB and RIP records. More details can be found in the description of the file *GENLIB in section MTS-280/27455.

Optional libraries are referenced by concatenating them to the end of the user specified source. For example, the contents of the file -LOAD might be loaded allowing reference to the user optional library MYLIB and the SSP library by the following RUN command.

```
$RUN -LOAD+MYLIB+*SSP  *SINK*
```

Pre-Defined Symbols and Low Core Symbol Dictionaries

The dynamic loader allows external symbols to be pre-defined at the beginning of the loading process. It will also search an "external symbol dictionary" for the definitions of external symbols which have been referenced but not yet defined whenever an LCS record is encountered. Both of these facilities are available through the LINK, LOAD, and XCTL subroutines. They are a convenient way to allow the programs loaded via LINK, LOAD, and XCTL to reference routines or data items which are already loaded. This allows the user to set up some semblance of an overlay structure.

Both the pre-defined symbol and "low-core symbol" capabilities require tables called external symbol dictionaries. An external symbol dictionary consists of (12*N+4) bytes, where N is the number of external symbols in the dictionary. The dictionary must begin on a full-word boundary. The first word of the dictionary contains N, the integer number of symbols entered in the dictionary. The rest of the table consists of external symbol entries, each being twelve bytes long. The first eight bytes of a symbol entry contain the EBCDIC name of the symbol, left-justified with

trailing blanks. The last four bytes of the entry contain the value of the symbol as a full-word integer.

As an example, suppose that a language scanner is to be broken into an overlay structure consisting of a scanner which links to a different module for each type of statement. Each statement module may in turn refer to the subroutines EXP, PUN, GEN, and NXTCHAR to scan the next expression, print on error message, generate output, and acquire the next character, respectively. An external symbol dictionary to define these symbols might be:

```
MYEXTSYM  DC  F'4'             Four Entries
          DC  CL8'EXP'          Defines EXP
          DC  A(EXPRTN)         Internal, different name
          DC  CL8'PUNT'         Define PUNT
          DC  A(PUNT)           Internal, same name
          DC  CL8'GEN'          Define GEN
          DC  V(GENRTN)         External, different name
          DC  CL8'NCTCHAR'      Define NCTCHAR
          DC  V(NXTCHAR)        External, same name
```

Notice that it matters not whether the symbol is external or internal in the routines already loaded. It may even have a different name in the program already loaded.

Now these symbols and the definitions given above might be passed on to a program to be loaded using LINK, LOAD, or XCTL via the second parameter to those subroutines. The symbols in the above table would then be appended to the symbols which MTS normally pre-defines and would become defined during the initialization of the loader. These symbols can then be referenced by the object modules being loaded. No object module can redefine any pre-defined symbols, however. So these external symbols are reserved symbols to the program being loaded.

The external symbol table constructed above could alternatively be used as a "low-core" symbol dictionary. The loader would search the external symbol dictionary for the definitions of external symbols which have been referenced but not yet defined whenever an LCS record referencing the "low-core" symbol dictionary was encountered. The advantage of this method is that the program being loaded can define its own symbols with the same names as some of the symbols in the low-core symbol dictionary without conflict – the program's own definition will be preferred.

In order to reference the low-core symbol dictionary, the loader must have an external symbol defined which is the base of the table. This can be accomplished by making the name of the table a pre-defined external symbol. For instance, in our example above we might have the second parameter of the call upon LINK, LOAD, or XCTL point to the following table:

```
PREDEF    DC  F'1'              One entry
          DC  CL8'MYEXTSYM'     Define MYEXTSYM
          DC  A(MYEXTSYM)
```

Then the occurrence of an LCS record referencing the symbol MYEXTSYM will cause each symbol contained in that external symbol dictionary to be defined with the value given in the dictionary if it has been referenced but not defined by the object modules loaded thus far. If the object modules loaded define any of the symbols in the table, the definition given by the modules will over-ride the dictionary definition.

It should be pointed out that the symbols defined as pre-defined symbols or in low-core symbol dictionaries may have as their values the addresses of any items in the loaded program which the user wants to reference in the program to be loaded. For instance, the address of a subroutine, a common section, or a variable might be passed on. For furthermore, the address given as the definition of an external symbol need not have the same name or even any name in the already loaded program. It is just an address which is associated with EBCDIC characters in the table to form an external symbol definition for the program to be loaded.

When an already loaded program calls LINK, LOAD, or XCTL to load another program, the loader starts fresh in the loading of the new program. None of the symbols in the loaded program are available to the program to be loaded, unless explicitly passed on via a pre-defined symbol or low-core symbol dictionary. Likewise, symbols in the program being loaded can be passed back to the already loaded program only via the fourth parameter for LOAD. Hence there is usually no problem of conflicting symbol names in programs which are loaded via LINK, LOAD, and XCTL.

INTERNAL SPECIFICATIONS

This major section describes the internal structure and workings of the MTS system. It is designed for the edification of all those interested and for the use of the systems programmers working on the system. No user should ever attempt to directly call any of the subroutines which might be listed in this section.

FILE AND DEVICE MANAGEMENT

This section is concerned with how files and devices are handled internally, at the level of a user of the file subroutines. The file subroutines are separately described in section MTS-350.

MTS-321-0

12-1-67

MTS INTERNAL SPECIFICATIONS

FILE AND DEVICE MANAGEMENT

INTRODUCTION

The basis of file and device management is the File or Device Usage Block (FDUB). This kind of beast is attached to logical functions, such as SCARDS, SPRINT, SPUNCH, AFD, SOURCE, etc. A given FDUB is pointed to by exactly one "logical device". It in turn "points", by means of an LDN if device and a FCB location if file, to an appropriate entry for the physical device it is fastened to. Since these information blocks for physical devices or files may have more than one FDUB pointing to them, each has a use count, so that it will be known when the device can be released or file closed.

The information block for a file is a File Control Block (FCB). Except for the use count, name, and the chain-pointer, the information in it is used only by the file manipulation subroutines.

The information block for devices consists of a number of parallel tables, indexed by LDN.

The general sequencing for an I/O call is as follows: the user CALLs (OS type I) a public entry (such as SCARDS) which goes to the the Device Support Routines Interface prefix (DSRI prefix), which chooses the appropriate Device Support Routine (DSR) and goes to it. The DSR returns to the Device Support Routines Interface postfix (DSRI postfix), which finally restores registers and RETURNs to the user. More specifically:

PUBLIC ENTRY

The Public entry saves registers in the save area, identifies what is the location of its FDUB, and whether the call is READ, WRITE, or other. It then transfers to the DSRI prefix.

DSRI PREFIX

DSRI prefix establishes necessary addressability and the location of DSECT. It checks FDUB location to see if is zero - if so this "logical device" (e.g. SCARDS,SPUNCH,etc.) was never set up and the user is prompted to either define it now or cancel. If the FDUB refers to a file and this file has not been opened an attempt is made to open it. If a file does not exist or a device is unavailable the user is prompted to enter a replacement or cancel. At the time SCARDS, SPRINT, SPUNCH, SERCOM were set up, the device type was checked to see if it was an input or output device

as required, and the input/output bit in the FDUB was set accordingly. The devices for READ and WRITE could not be checked then, so they are checked now to make sure that it is an input type device for READ and output type device for WRITE, and the FDUB input/output bit is set appropriately. On the basis of the FDUB input/output bit, the appropriate subroutine is selected from the transfer vector pointed to by the FDUBSBR field of the FDUB. The third argument of the call (modifiers) is tested to see if present and if so, the modifiers from the call and the default modifiers in the FDUB are appropriately combined and placed in the rightmost three bytes of the FDUBS field of the FDUB. If the call is for output and the "return line number" bit is off or the "trim" bit is on, a fake parameter list is constructed which is a copy of the user's but with the line length moved to the MTS DSECT so it can be modified for the "trim" bit and with a fourth argument if the call was sequential with the "return line number" bit off. (since the user is not required to supply such in this case).

The FDUBCL region of the FDUB is of main use in sequential access: the user does not supply a number, so the one in the FDUB is kept as a current one and is updated. Hence for a sequential call, the FDUBCL is added to the FDUBIL field and the result placed where the user's calling sequence specifies the line number should go.

If output and case conversion is wanted, it is done now, to the line the user's calling sequence indicates.

A prefix character is set up, either the single character from PREFIXC or a line-number (from user's calling sequence), according as the prefix bit is not or is set, respectively.

The DSR error bit in FDUBS is set to zero.

Finally, if the attention latch is not set (indicating a non-immediate attention has come through since the last I/O operation), control is transferred to the DSR.

DSR

DSR does the actual I/O operation requested. If the peel modifier bit is set (input), a line number is peeled off the front of the line and returned into the user's argument list. Control is transferred to the DSRI postfix.

DSRI_POSTFIX

If the DSR error bit is 1, the message whose length is in GR2 and location is in GR3 is typed out and the return is set to INLOOP unless SETIOERR has been called. DSRI postfix takes the line number from the user's parameter list and puts it in FDUBCL. If this was an input call and case conversion was wanted, it is done now. Unless the implicit-concatenation-off bit is set, the line (if input) is checked for "$CONTINUE WITH FDname" which signifies a change of file or device for this logical input device. If the return code is 4 or the ending line number is exceeded and this FDUB is not the last one in a concatenation the next FDUB is set up. Finally, the registers (saved at the public entry) are restored and a return is made. The return code was set up by the DSR.

File or Device Usage Block DSECT Definition

```
FDBDCT    DSECT
FDUBS     DS    F           SWITCHES - INCLUDES "MODIFIERS"
*                           BYTE 1: BIT 0=0 DEVICE
*                                     1 FILE
*                                BIT 1=0 INPUT
*                                     1 OUTPUT
*                                BIT 2=0 NO INDEXED OP
*                                    =1 INDEXED OP
*                                BIT 3=0 NO DSR ERROR
*                                    =1 DSR ERROR
*                                BIT 4=0 FILE NOT OPEN
*                                    =1 FILE IS OPEN
*                                BIT 5=0 FDUB IS OK
*                                    =1 FDUB IS BAD
*                           BYTES 2,3,4: RESULTANT MODIFIER BITS
FDUBLN    DS    F           FCB LOCN OR LDN
FDUBBL    DS    F           BEGINNING LINE NUMBER
FDUBCL    DS    F           CURRENT LINE NUMBER
FDUBEL    DS    F           ENDING LINE NUMBER
FDUBIL    DS    F           INCREMENT
FDUBSBR   DS    F           SUBROUTINE LOCATION
FDUBSCR   DS    F           SCRATCH (SEQUENTIAL POINTER)
FDUBNAM   DS    F           BACK POINTER TO WHO'S USING IT
FDUBDS    DS    F           DEFAULT SWITCHES
FDUBCH    DS    F           CHAIN TO NEXT (FOR CONCAT FILES)
*
FDUBL     EQU   *-FDBDCT    LENGTH
FDUBAL    EQU   FDUBL+1*4   ACTUAL LENGTH-INCLUDING CHAIN LINK
```

MTS - INTERNAL SPECIFICATIONS

FILE AND DEVICE MANAGEMENT

STRUCTURE OF DEVICE TABLES

These tables are indexed by LDN

1) DEVTBLS 1 word/entry

Contains switches

INUSE Byte 1 x'01' - In use bit

OPPEND x'02' - Operation pending on this device

2) DEVTBLT 1 word/entry

Contains TYPE of device [4 char BCD]

3) DEVTBLN 1 word/entry

Contains NAME of device [4 char BCD]

4) DEVTBLU 1 word/entry

Use count for device in question.

5) DEVTBLC 1 word/entry

Header for lists of space allocated by device routines. All storage allocated by DSR should be chained together by pointers

in the first word of each block so it can be automatically released when the device is released. All storage on the chain must be system level storage.

6) DEVTBLP 4 double words/entry

Contains parameters for device.

First four words intended as parameter region for unit check routines. Last four words provide space for two CCW's.

DEVICE SUPPORT ROUTINES (DSR) - SPECIFICATIONS

With the idea that the code for most devices will not be permanently resident, but will be brought into a buffer as needed, the following structure is set forth.

All of the routines to service a particular device will occupy a single file. The "entry point" of this file (probably - but not necessarily - the first location) will be the location of a list of adcons, each pointing to the entry for the particular subroutine in question. These are, in the order they will appear:

1. INITIALIZATION
2. DITCH
3. GETFROM
4. WRITEON
5. SET ATTENTION
6. WAITFOR
7. RELEASE

[This list is not necessarily complete.]

The description of each of these follows.

Common Information

Each will be called as follows:

GR0 will contain FDUB location.

GR4 will contain the DSECT pointer.

GR5 will point to a list of adcons, the first of which points to the subroutine LINNBR which peels line numbers off the front of a line and converts them. The third word in the list of adcons, points to a list of 6 full words giving the offset from the front of the DSECT of the 6 device tables in the order they are described elsewhere.

GR11,12 will have the appropriate bases.

GR13 will point to a save area into which the appropriate save has been done. This area may be accessed but not changed.

GR14 will contain the return location.

GR15 will contain the location of the first location in the subroutine.

Each should return as follows:

(a) For GETFROM and WRITEON do not restore registers from save area. Do BR 14 with GR0,GR1,GR4,GR10-14 the same as they were when the routine was called

(b) Others - standard OS return.

For both cases GR15 should contain the return code.

1. INITIALIZATION

On entry, in addition to what is specified above, GR1 will contain a location to transfer to if the device routines should decide at any time the person is to be signed off. [Appropriate closing out will be done at this time, and then the DITCH entry will be called. Calls to the other entries preceding the DITCH call may be ignored.]

2. DITCH

On this entry, everything will have been released and/or closed out except the master source/master sink device(s) and any vital storage associated with them, such as FDUB, FCB, etc. This is called only for master source. This subroutine is responsible for making sure, in the appropriate way, that the current user has been removed from that device. When a return from this subroutine is effected, it signifies a no-user condition.

3. GETFROM

On entry, in addition to what was specified above, GR1 will contain the location of a parameter list, GR2 will contain the location of prefix character(s) to be output at the front of the line, GR3 will contain the number of prefix characters. The modifiers provided by the user as the third argument in the parameter list will have been combined with the ones

following the FDname and the resultant directive bits placed in the FDUBS field of the FDUB. For each pair of associated modifier bits, exactly one of these bits will be on. Hence bit testing for modifiers should be done in FDUBS and not in the parameter list. Testing and appropriate action taking for the prefix bits, trim bits and the case conversion bits are done in the DSRI; the DSR is responsible for acting on the rest.

Line numbers are for the most part handled by the DSRI. The only case that the GETFROM DSR must handle is if the "peel" modifier or bit is set. In this case the routine must peel off the line number, convert it via LINNBR, and return it to the user's parameter area. In no case should the DSR change the line number information in the FDUB; this is taken care of by the DSRI.

The parameter list is described in MTS-252. The subroutine shall obtain the requested data and then return. The return code should be zero for normal return, and 4 for EOF return.

If an error is detected, the location of an error comment should be placed in GR3, it's length in GR2, a return code >4 in GR15, and the DSR error bit should be set in FDUBS.

4. WRITEON

On entry, in addition to what was specified above, GR1 will contain the location of a parameter list, GR2 will contain the location of prefix character(s) to be output at the front of the line, GR3 will contain the number of prefix characters. The parameter list is described in MTS-252. The subroutine shall output the data as specified and return. The return code should be zero for a normal return.

If an error is detected, the location of an error comment should be placed in GR3, it's length in GR2, a return code >4 in GR15, and the DSR error bit should be set in FDUBS.

5. ATTENTION

This is called only for the master source. On entry, in addition to what was specified above, GR1 will contain a zero or location that can be called if an attention or pseudo-attention (from full-duplex devices) occurs. If GR1 contains zero then all further attention interrupts should be inhibited. This call should be made with DSECT and base registers set up, and with reg 0 containing the location at which the interrupt occurred or zero if unknown in which case the attention will be processed later. Also register 13 should point to a save area and register 1 to an area containing the general registers (in the order 0 to 15) at the time of the interrupt. The occurrence of an "attention" should inhibit further of the

same. If a return is made (before another set attention call), this means enable further attentions (RC=0) or inhibit further attentions (RC=4), and return to what was interrupted.

6. WAITFOR

This subroutine is responsible for waiting, in the appropriate way, for the next user to appear at the given device. When a return from this subroutine is effected, it signifies a new user has been sensed and things should start up again. This is normally called (for MSOURCE only) upon initial start-up, to wait for the first user, and thereafter after a DITCH call which got rid of the previous user. If the device is capable of producing an answer back then when WAITFOR returns register 1 should contain the location of a one byte count followed by the answer-back.

7. RELEASE

This is called only for devices that are not the master source. It is the last subroutine called before the device is released, and should make sure appropriate close-out action is taken. It should neither release the device nor the storage associated with it which shall have been put on the DEVTBLC chain.

PROCESSOR INTERNAL SPECIFICATIONS

This section contains descriptions of the internal workings of various processors in the system.

LOADER INTERNAL SPECIFICATIONS

Introduction

The dynamic loader itself is a small, re-entrant, system-independent, resident subroutine which processes loader input records, producing the linked program in core storage and printed output. The loader itself does not perform input/output operations or acquire storage for the control sections it loads; all this must be provided by the routine which calls upon the loader. Because such problems are left to the system in which the loader resides, the overall flow of the loading process as seen by the user is more dependent upon that system than upon the loader itself. It can be said that any similarity between the loader and the loading process as seen by the user is purely coincidental.

The loader is a subroutine which may be called several times to accomplish one loading process. Whenever the loader returns to the routine which called it, it makes available to that routine a status word which indicates the exact status of the loading operation at that time. If some error condition existed which the calling routine could correct, it could correct the condition and call the loader again to continue loading. Hence it is possible for control to bounce back-and-forth during the load process until finally all the smoke clears and the program is loaded.

This manual shall assume that all other loader manuals and the writeups for the $RUN and $LOAD commands and LINK,LOAD,and XCTL subroutines have already been read. The next few sections of this writeup will describe the loader as a subroutine. No attempt will be made to explain the interaction of the loader with MTS until the subroutine description has been completed.

Name

The name of the resident subroutine which accomplishes the loading process is UMLOAD. This name appears in the map of resident system symbols.

Function

The function of UMLOAD is to link and load one or more object modules, each consisting of one or more control sections, leaving the executable program in a specified region of core. The loading procedure is done in an "interactive" mode with the calling program. Input/output routines must be provided by the calling program.

Calling Sequence

the calling sequence for UMLOAD is a standard OS/360-Type 1 calling sequence, with the exception of the save area. The so called save area is a large area which must contain its length in its first four bytes. The front of this area is used as a standard save area. The remainder of it is used as the psect (changeable storage) for UMLOAD. Re-entrant code and all that you know. The register conventions are:

GR01: Address of the first element of a parameter list. The parameter list is explained in detail below.

GR13: address of the first word of a region of core which will be used by the loader as a save area and psect. The beginning of this region must be set up as an OS/360 save area. The first word of this region must contain the length of the region in bytes. This region must be at least 728 bytes long. Actually, to make the loader internal tables of anywhere near usable size this area should be at least 4096 bytes in length.

GR14: Address of the return location in the calling program.

GR15: Address of the entry point of the load subroutine (UMLOAD).

Parameter List

The parameter list for UMLOAD is a standard OS/360 parameter list, with the exception that many of the parameters which themselves are addresses are contained in the parameter list instead of being pointed to by elements of the parameter list. Hence it would be a little difficult to write a call upon UMLOAD in a compiler language. An explanation of each parameter follows:

PAR1: The full-word address of a subroutine which can be called to read the next input record. The calling sequence for this subroutine is assumed to be identical to the calling sequence for the system subroutine SCARDS.

PAR2: The full-word address of a subroutine which can be called to write the next output record. The calling sequence for this subroutine is assumed to be identical to the calling sequence for the system subroutine SPRINT.

PAR3: The full-word address of the first word of a region of core in which the program will be loaded. The first word of this region must contain the length of the region in bytes. It is possible to continue loading with a new region if the last region was not long enough to hold the remainder of the program. This capability will be discussed later in the section on error recovery.

PAR4: The full-word address of the origin of an initial external symbol dictionary. This feature allows external symbols to be pre-defined before the actual loading operation takes place. If this parameter is zero no symbols will be pre-defined. The format of this initial external symbol dictionary will be discussed below.

PAR5: The full-word address of a loading status word. The first three bytes of the loading status word must be set to zero before the first entry to the load subroutine. Upon a return from the load subroutine these bytes will have bits set to indicate the exact status of the loading procedure. The loading procedure can usually be continued by correcting any indicated errors and calling the loader with these bytes of the loading status word unchanged. The last byte of the loading status word can be set by the calling program to control the amount of printed output generated by the loader. The meaning of the bits within the loading status word will be discussed below.

PAR6: The full-word address of a full-word in which the load subroutine will return the entry point of the loaded program. The entry point is determined in the standard manner from information on ESD,TXT and LDT records.

PAR7: The full-word address of a full-word in which the load subroutine will return the number of bytes used in the current load buffer. If this address is zero, the value will not be returned. The load buffer is the region pointed to by the third parameter.

PAR8: The full-word address of the first word of a region of core in which the current external symbol dictionary will be returned. If this address is zero, the external symbol dictionary will not be returned.

PAR9: The full-word address of a full-word in which the load subroutine will return the length of the next control section to be loaded if the loading procedure is terminated by a full load buffer. If this address is zero, this value will not be returned. Loading can be continued by calling the load subroutine with parameter three pointing to a new region of core which is at least as long as the next control section.

Return Sequence

The return sequence for UMLOAD is a standard OS/360 Type 1 return sequence. The register conventions used are:

GR0-14: Restored to original contents during call.

GR15: A return code indicating the severity of any detected errors:

0 Normal return. No errors were encountered.

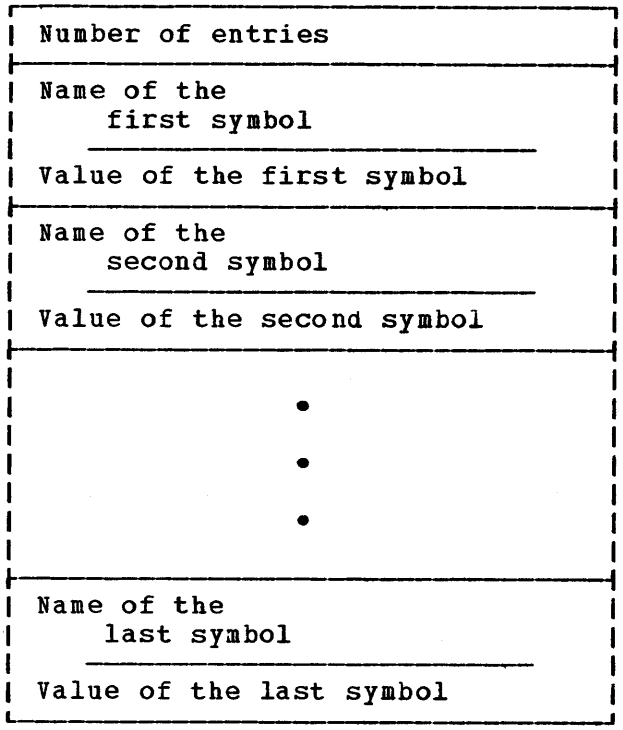
4 Non-fatal error return. Errors were encountered, but the program can probably be executed as loaded.

8 Fatal error return. Severe errors were encountered during loading. The program probably is not capable of being executed successfully.

External Symbol Dictionary Format

A program which calls UMLOAD can gain access to or cause to be altered the external symbol dictionary stored within the loader's psect in three ways. First, the program can pre-define external symbols by having as parameter four the address of an initial external symbol dictionary. Second, the program can obtain the current definitions of all symbols at the time of load termination by having parameter eight point to the area in which the loader should leave a current external symbol dictionary. Finally, symbols which have been referenced but are not yet defined can be defined from an external symbol dictionary pointed to by an LCS record. All of these dictionaries have identical formats. It should be remembered though, that this is not the format of the table used internally by the loader. It is the format of an external table through which information can be passed back and forth between the loader and the calling program.

An external symbol dictionary consists of (12*N+4) bytes, where N is the number of external symbols in the dictionary. The dictionary must begin on a full-word boundary. The first word of the dictionary contains N, the integer number of symbols entered in the dictionary. The rest of the table consists of external symbol entries, each being twelve bytes long. The first eight bytes of a symbol entry contain the EBCDIC name of the symbol, left justified with trailing blanks. The last four bytes of the entry contain the value of the symbol as a full-word integer. An undefined symbol has 000000 as its value.



External Symbol Dictionary Format

Error Recovery and Restart Procedures

The load subroutine has been designed to allow the load process to be continued, perhaps with some corrections, by calling upon the loader again after it has already returned from loading. This is accomplished by presenting the loader with the save area (psect) it has used up to the present point in the loading process and a loading status word with the first three bytes identical to those in the one it last returned. It is essential that the return code and loading status word returned by the loader be understood to take advantage of this continuation procedure. For this reason, both of these are described in more detail below.

Return Codes

1. Normal return code

A normal return code indicates that as far as can be determined, the load process has been completed successfully and the program is ready for execution. The load process was terminated by either an LDT record or an END-OF-FILE return from the input routine. Loading can be continued if desired; for instance, to link another control section to all that has been loaded already, but it is very unlikely that anyone would want to do something like this. The loading status word will have only bit 1 on to indicate that some loading has been accomplished.

2. Non-fatal error return code

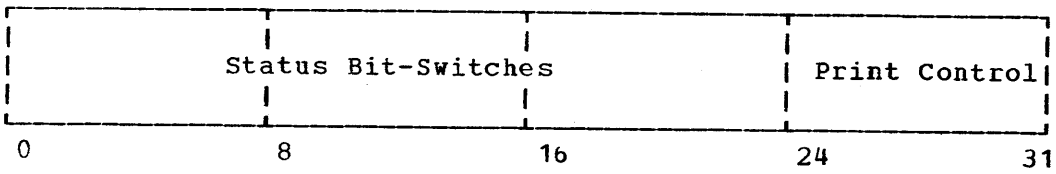
A non-fatal error return code indicates that some errors have been detected during the load process, but they probably aren't serious enough to prevent the program from being executed. In some cases, this assumption may not be valid. The loading status word will have bit 1 on to indicate that some loading has been accomplished, and will have other bits on to indicate the particular errors detected. Loading can always be continued, if desired, after a non-fatal error return code.

3. Fatal error return code

A fatal error return code indicates that at least one error has been detected during the load process which is serious enough to jeapordize any meaningful execution of the program for the moment. It is possible that some correction can be made and loading continued, although not all fatal errors can be corrected without completely restarting the load process. An undefined symbol is an example of a fatal error which can be corrected by continuing loading, while overflowing the loader internal tables is an example of a fatal error which can be corrected only by restarting the load procedure from the beginning. Bits within the loading status word are set much like for non-fatal errors, with the exception that bit 0 may be on and if on indicates that loading cannot be continued.

Loading Status Word Format

The loading status word is broken into two parts. The first three bytes are bit-control switches which indicate the current status of the load process. These bits must all be zero when the loader is first called to begin loading, and must not be changed if the loader is called to continue a previously started load. The last byte of the loading status word is used to control the amount of printed output produced by the loader. This byte can be changed by the calling program at any time to vary the amount of output. Both of these parts of the loading status word are described in more detail below.



Format of the Loading Status Word

1. Status Bit-Switches

The first three bytes of the loading status word are bit-switches which indicate the current status of the load process. The following list describes the meaning of an <u>on</u> condition for each bit.

BIT-0 A non-recoverable error has been detected. Loading cannot be continued. Loading can be restarted, however, by correcting the cause of the error, re-positioning the input file at its beginning, and calling the load subroutine with the first three bytes of the loading status word set to zero.

BIT-1 This bit is always turned on when the loader returns to the calling program. It simply indicates that some loading has been accomplished. It is used by the loader to determine if loading is being continued or started from scratch.

BIT-2 The next control section to be loaded is longer than the available space remaining in the current load buffer. This is a fatal although recoverable error. The length of this next control section is returned via parameter nine if that parameter address is non-zero. The number of bytes actually used in the current load buffer is returned as usual via parameter seven if that parameter address is non-zero. It is likely that some space at the end of the load buffer has not been utilized and can be used for some other purpose. If the first control section encountered was longer than the load buffer no space in the buffer will have been used and the entire buffer can be reused for some other purpose. Loading can be continued by calling the load subroutine with parameter three pointing to a new load buffer at least as long as this next control section. The input file must not be positioned before recalling the loader if loading is to be continued.

BIT-3 The last input record read was longer than 256 bytes. This is a fatal and non-recoverable error.

BIT-4 The last input record read was not a recognizable loader input record. This is a non-fatal error and loading has been terminated as if this record was an end-of-file. The contents of the guilty record are printed with the loader output.

BIT-5 A load map has been started in the loader printed output but has not yet been completed. This is used by the loader to prevent spurious maps on multiple calls and returns from the loader. This is only an internal switch of the loader. It does not indicate that an error has occurred.

BIT-6 At least one external symbol has been multiply-defined. This is a non-fatal error. The first definition of each multiply-defined symbol has been used.

BIT-7 The external symbol dictionary/relocation dictionary table has overflowed the area available to it. This is a fatal and non-recoverable error. The error can be corrected only by restarting the loader either with a larger psect (save area) or with a re-ordered input deck which requires less table space.

BIT-8 The external symbol identifier (ESID) which specifies the symbol whose relocation factor is to be used in the current address computation is either not defined or the associated symbol does not have a relocation factor. This is a fatal and non-recoverable error. Several causes of this type of error have been put forth in section MTS-296.

BIT-9 The external symbol identifier (ESID) which specifies the symbol whose value is required in the next address computation is undefined. This is a fatal and non-recoverable error. This error is caused by the same types of conditions which cause bit 8 to be set.

BIT-10 An external symbol which is referred to in the program has not been defined. This is a fatal although recoverable error. The address constants which refer to this symbol have not been altered. If loading is continued and the undefined symbol is subsequently defined, all references to it will be rectified correctly.

BIT-11
•
•
•
BIT-23 These bits are not in use yet.

2. Print Control

The last byte of the loading status word is used to control the amount of output printed by the loader in the following way. Every type of printed output which can be produced by the loader belongs to a certain class of printed output. Every class, with the exception of the class for fatal error messages, has associated with it a bit in the fourth byte of the loading status word. An output line is printed only if the bit associated with its class is turned on. Of course, fatal error messages are always printed. The class numbers have been assigned so that if the class numbers for all classes to be printed are added up and the fourth byte of the loading status word is given this value, the desired print control is in effect. A list of the class numbers, the associated loading status word bits, and a description of each class of output follows:

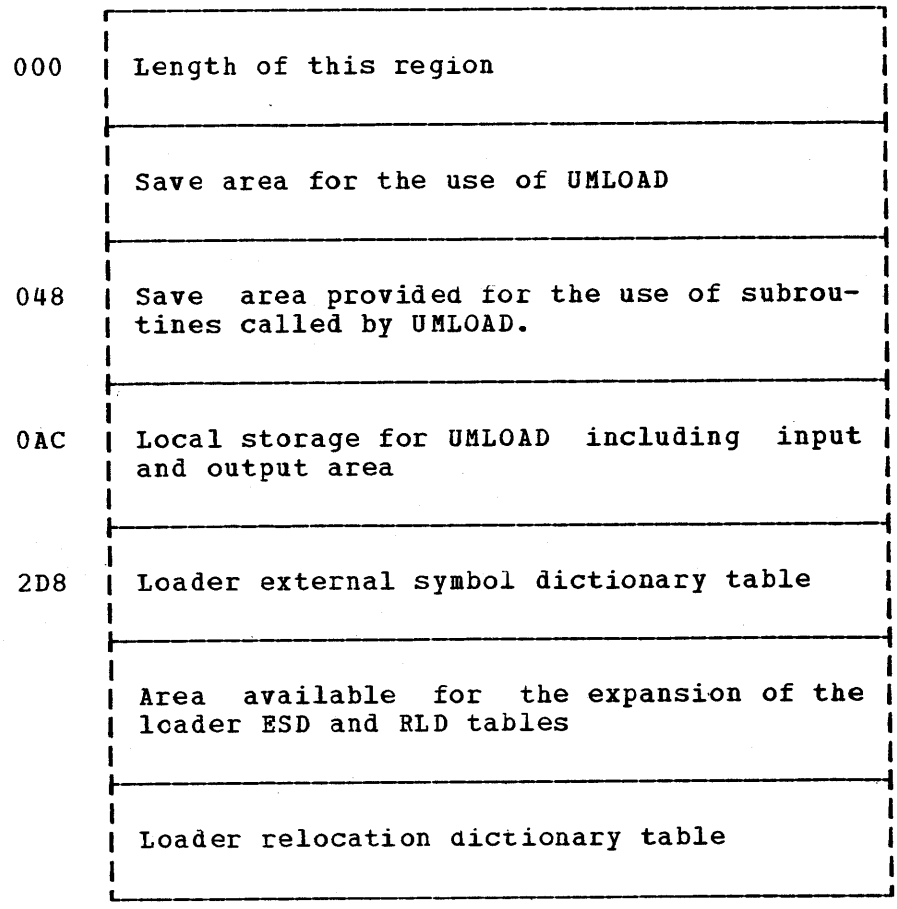
CLASS	BIT	DESCRIPTION
0	--	All fatal error messages except the error message for undefined symbols and the error message for a load buffer overflow.
1	31	Non-fatal error messages. These can be considered warnings of possible error conditions.
2	30	Information lines, such as the line giving the entry point address.
4	29	Positioning or format lines, such as blank lines or the lines of dots.
8	28	Symbol reference lines. These lines, if printed, form a reference table for all of the external symbols, giving the assigned core address of every adcon referencing the external symbols.
16	27	The error message stating that the load buffer has overflowed.
32	26	The error message stating that a symbol is undefined.

Output classes 16 and 32 were created so that errors which are

normally corrected automatically by the program calling the loader can have their error messages turned off.

General Organization of the Loader Psect

When the load subroutine is called, it must be provided with an extra-long save area with the length of the area stored in the first word of it. This save area is then used as the psect (changeable storage) for the loader. This allows the loader to be re-entrant.
The general organization of the loader psect is as follows:



Organization of the loader psect.

Note that the loader begins its external symbol dictionary table righ after the fixed-length information within its psect. Furthermore, i builds its relocation dictionary table starting at the end of the psect, a defined by the length in the first word of the psect. Hence both table may grow into the unused space between them. If the tables do eve overlap, the loader returns with a fatal and non-recoverable error.

More Details on the Loader Structure

Many details of the loader have not yet been presented. For instance, the structure of the loader's internal tables and the steps taken t process each type of loader input records has not yet been presented. These items will not be presented in this manual. Anyone interested i details such as these can find them in the loader assembly listing, as i is well-documented with many comments and paragraphs of explanation.

The Loader-MTS Interface

As has been stated before, the details of the loading process as seen by the user depend to a great extent upon the manner in which the system uses the loader subroutine. Following are just a few of the more pertinent details concerning the interface between MTS and the loader.

1. The first call upon the loader is made with a program or load buffer of length zero. This forces the loader to make a fatal error return with a program buffer overflow error when the first control section is encountered. MTS then calls the loader providing it with a buffer exactly as long as the first control section. The loader then can load the current control section in this buffer, but will have to repeat the process outlined above when it encounters the next control section. This forces each control section to be loaded into a separate and independent area of core storage.

2. If the loader returns with the fatal error indicating that there are undefined symbols, MTS then calls the loader giving it an input subroutine which reads the system library file. Hence undefined symbols are searched for in the library.

3. A small number of symbols are pre-defined by MTS. These symbols cannot be redefined by a user.

4. The setting of the print control byte of the loading status word by MTS is a colorful process designed to allow or suppress such things as error messages, maps, and dotted lines and depends upon whether input is from the library or user specified source and whether a is desired or not. The general processing is:

A. Set the print control and load the user specified input.
B. Reset the print control.
C. If symbols remain undefined load the system library.
D. If no symbols are undefined and a map was requested load the contents of *DUMMY* to get the map.
E. If no errors and execution desired go to it.

The setting of the print control byte in the various states is given in the following table:

Loader input source	Map	No Map
User specified source	5	1
*LIBRARY or *DUMMY*	39	33

FILE ROUTINES

This section describes the internal structure of a line file, and gives the calling sequences for the internally used subroutines to handle these files.

FILE FORMAT - GENERAL DESCRIPTION

Allocation of space and cataloging

Space allocated to files, and the existence of them, is recorded in an IBM-type VTOC. All information in and about the VTOC is maintained in standard OS format with the exception of "TYPE 5" records (available space) which are not quite correct. All routines using the disk use the standard system subroutines to allocate and release space for files, as well as for creating and destroying the files themselves.

Files written through system subroutines (SCARDS, SPUNCH, etc)

Files written through these subroutines, and maintained through them, are kept in a format conducive to easy updating by line number. Thus every line in such a file must have a line number associated with it. Such files are also limited to a size of 218 records on the disk, the implications of which and reasons for which will become clear later.

Each such file consists of three logical components, which will hereafter be referred to, with some consistency, as the track index, line directory, and line file.

The track index is used simply to associate a physical disk address (disk pack name, cylinder, track, and record address) with a logical record number (integer between 1 and 255) which is used internally. There is a one-one correspondence (hopefully) which associates a single logical record number with each of the physical records allocated to the file.

The line directory indicates where each line of the file is, as a logical record address, offset within that record, and how long the line is. This directory consists of fixed length entries, ordered by line number, so that it can easily be searched and modified. It only points to the lines themselves, which are resident in:

The line file contains the actual EBCDIC (or whatever) lines in the file. This file is unordered, and what pieces are used or unused is recorded only in the line directory. The lines in the line file may be of variable length, there may be unused 'holes' in the middle of or at the end of tracks, and it is in general useless without the line directory to make sense out of it.

Physical format of the components

THE TRACK INDEX

Each entry in the track index is 16 bytes long, and there is one for every physical record allocated to this file. The entries have the following format:

BYTES 0-5: Name of disk pack the record is on

BYTES 6-7: Physical cylinder address

BYTES 8-9: Physical track address

BYTE 10 : Physical record address (always 1 for a 2311, 1 or 2 for a 2314)

BYTES 11-15: Unused

The logical record number associated with a physical record entry is determined by the position of the entry in the track index, in the FORTRAN sense, which is to say that the first entry in the track index is logical track number 1, the second is number 2, and so on.

THE LINE DIRECTORY

Each entry in the line directory is 8 bytes long, and there is an entry for every line in the file, and also entries for

1) All the 'holes' or unused pieces of tracks that are partially used

2) The as-yet-unused space at the end of the file.

Entries in the line directory have the following format:

BYTES 0-3: The number (as a 32-bit signed integer) of the line which this entry describes (see below for special 'hole' entries)

BYTE 4: Logical record number of the track where the line itself resides.

BYTE 5: Character (byte) count for the line, i.e., how long it is.

BYTES 6-7: Offset, indicating how far from the beginning of the

record indicated in byte 4 this line begins. (Offset is in bytes).

The entries in the line directory are ordered by line number, and 'hole' and 'end-of-file' entries are given the highest possible line numbers so that they will naturally appear at the end of the directory. 'Hole' entries look just like normal entries except that they have a line number of hex '7FFFFFFE'. Such entries indicate that there is an unused piece of line file at the place and of the size indicated in the second four bytes of the entry. The entry indicating where the file ends (beyond which, all space is available) has a line number of '7FFFFFFF' (it is always the last entry in the directory), and in this entry byte 5 is always 0, and bytes 6-7 indicate the next available byte on the record (it and all succeeding bytes are presently unused). All logical records beyond the one indicated in byte 4 (which is the one being filled) are also unused.

THE LINE FILE

The line file may have all sorts of garbage in it, as when new lines are added to the file they are simply placed on top of whatever was there, and the deletion of lines is recorded only in the line directory, the line itself, or pieces of it, are left around in the line file until the space is reused.

How the components are tied together

Since all three of the components must exist within the area assigned to a single file, it is necessary to find where each of them is. This is easy for the track index, because it is always in the first record of the first extent assigned to the file, and begins at the eighth byte of the record The length, in bytes, of the track index is in bytes 0-1 of the record.

We must now make a distinction between "normal" and "extended" files.

A normal file is small enough so that both the line directory and track index may be contained in the first record (logical record #1). If this is the case, the line directory immediately follows the track index, and its length, in bytes, is given in bytes 2-3 of the record.

If the file is too big to be so cozy, then only the track index (TI) will be in track #1, and the line directory (LD) will be elsewhere. But where, you say? Ah-ha. The answer to that lies in bytes 4-7 of record #1. In any case where the TI and LD cannot exist together on record #1, the TI alone will be on record #1, and will point (as will be seen), to the first LD record , and the LD records (if there are more than one) will be linked together, all through bytes 4-7 of each LD or TI record , which are defined as follows:

BYTE 4: LOGICAL RECORD # OF PREVIOUS RECORD IN TI-LD COMBINATION (FOR TI THIS WILL BE 0)

BYTE 4: LOGICAL RECORD # OF NEXT RECORD IN TI-LD COMBINATIO
(FOR LAST RECORD THIS IS 0)

BYTE 6: LAST USED RECORD OF THOSE ASSIGNED TO FILE. THIS I
SIGNIFICANT ONLY IF BYTE 5 = 0.

BYTE 7: SELF-IDENTIFIER. LOGICAL RECORD # OF THIS RECORD .

When the TI and LD are not entirely contained in a single record, w
speak of the file as being extended. Record #1 contains the TI only (an
bytes 2-3 =0), and byte 5 points to the first record of the LD (which ma
point to further records in the LD), and in each record of the LD, byte
0-1=8 (the header length) and bytes 2-3= k, the number of bytes of the L
contained in this track.

File size limitations

1) The TI must be contained in a single record (3520 bytes), so a fil
cannot be assigned more 218 records.

2) A file may not have more than 16 non-contiguous blocks of track
(extents) assigned to it.

3) No line in the file may be longer than 255 bytes.

4) A file may be assigned space on only a single disk pack (thi
restriction may too may go away)

FILE OPERATION SUBROUTINES

FILE CONTROL BLOCK DESCRIPTION

The following table describes the file control block in such a way that one could generate either a DSECT or set of EQU's to reference it in ones program.

```
FCBN    DS    4F     ALLOW SIXTEEN BYTES FOR ALPHABETIC FILE NAME
FCBLK   DS    F      LINK TO NEXT FCB, OR ZERO
FCBCS   DS    F      CHECKSUM FOR FILE CONTROL BLOCK
FCBVC   DS    6X     6-BYTE NAME OF VTOC VOLUME
FCBCC   DS    H      2-BYTE CYLINDER ADDRESS OF VTOC
FCBTC   DS    H      TRACK ADDRESS FOR VTOC ENTRY
FCBRC   DS    X      RECORD NUMBER OF VTOC ENTRY
FCBBC   DS    X      COUNT OF ASSIGNED BUFFERS
FCBAI   DS    X      ACCESS INFORMATION (SUCH AS 'READ ONLY').
FCBEC   DS    X      COUNT OF EXTENTS ASSIGNED TO THE FILE.
FCBVF   DS    6X     VOLUME NAME FOR LINE DIRECTORY
FCBCF   DS    C      CYLINDER ADDRESS FOR FILE
FCBRF   DS    C      RECORD NUMBER FOR FILE.
FCBTF   DS    H      TRACK ADDRESS FOR FILE

 ****   FOLLOWING 16BYTES REPEATED UP TO BUFFER COUNT

FCBFB   DS    C      FLAG BYTE INDICATING BUFFER STATUS
FCBBA   DS    3C     MEMORY ADDRESS OF BUFFER
FCBBL   DS    H      LENGTH OF ASSIGNED BUFFER
FCBBV   DS    6C     NAME OF VOLUME RECORD NOW HERE
FCBBC   DS    H      CYLINDER NUMBER OF RECORD IN BUFFER
FCBBR   DS    C      RECORD NUMBER OF RECORD NOW IN BUFFER
FCBBT   DS    C      track ADDRESS OF RECORD NOW IN BUFFER
```

The size of a file control block is dependent on the number of buffers allocated to the file. In general the block size must be at least 48+16N bytes, where N is the number of buffers assigned to the block. Normally there will be need for only two buffers per block, so the block should be at least 80 bytes in length.

The following section describes the file control block in terms of EQU's.

```
FCBN   EQU 0
```

```
FCBLK   EQU 16
FCBCS   EQU 20
FCBVC   EQU 24
FCBCC   EQU 30
FCBCT   EQU 32
FCBRC   EQU 34
FCBBC   EQU 35
FCBAI   EQU 36
FCBEC   EQU 37
FCBVF   EQU 38
FCBCF   EQU 44
FCBRF   EQU 45
FCBTF   EQU 46
*****
FCBFB   EQU 48
FCBBA   EQU 49
FCBBL   EQU 52
FCBBV   EQU 54
FCBAC   EQU 60
FCBBR   EQU 61
FCBBT   EQU 62
```

The flag byte (FCBFB) is used to describe the status of each buffer assigned to the file. Each bit has the following meaning-

```
BIT    7   1 Indicates buffer in use
BIT    6   1 Indicates buffer must be written
BIT    5   1 Indicates buffer not standard
BIT    4   Not used
BIT    3   Indicates record has never been written
BITS 0-2   Priority of record in buffer
```

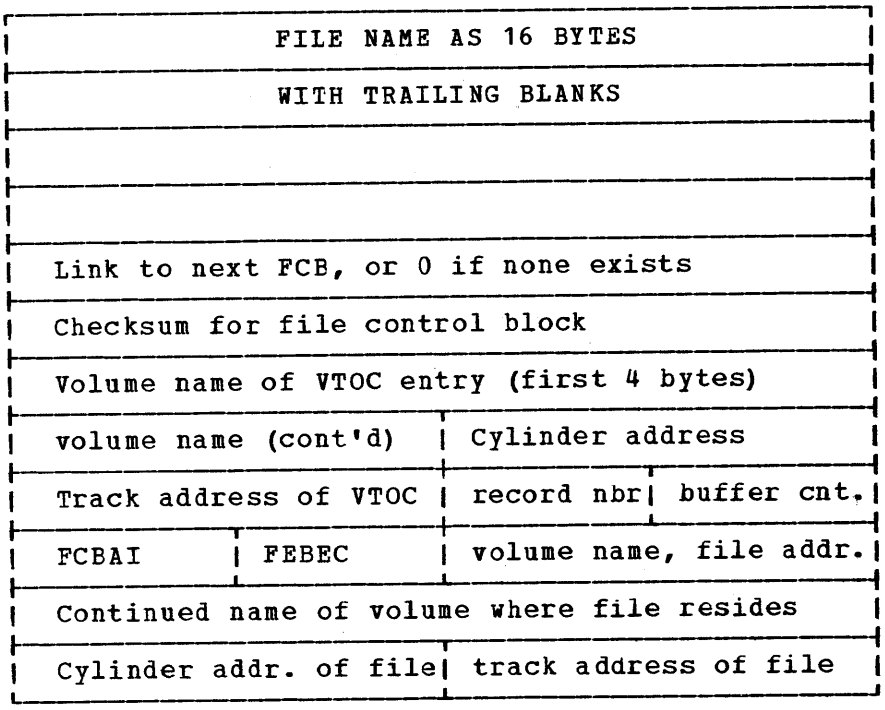
'Non-standard' means that the file system will make no attempt to manage the data in the buffer, and will not use that particular buffer in the course of normal operation, and will not close the buffer if such a call is executed.

If the user makes changes to the file control block (such as adding a new buffer) he should re-compute the checksum, or call 'CHKSUM' to do the job. He should also be careful to maintain the 'BUFFER COUNT' byte correctly.

MTS-352-0

12-1-67

FCB layout:



	BUFFERS #1	#2	#3
Flag byte | address of assigned buffer	48	64	80
Length of buffer | volume no. in buffer	52	68	84
continued name of volume for record in buffer	56	72	88
cylinder and track address presently in buffer	60	76	92

•
•
•(Additional 16 byte entries up to buffer count)
•
•

External File System Subroutines

GENERAL INFORMATION

As will be quickly obvious, these subroutines do not fit into any OS category, and in fact are not even consistent in their behavior amongst themselves.

All require the address of a working area (not an OS type save area) in GR13, and said area must appear on a double-word boundary.

Where parameter lists are needed, they also must be on full-word boundaries. Throughout these writeups (to avoid confusion) the words 'address' and 'pointer' are used synonymously and randomly, to refer to a 24-bit absolute memory address.

The file subroutines all save and restore all the registers except GR0 and GR15. GR15 always has a return code when exit is taken from the subroutine, and GR0 is used in device management. The file subroutines all assume logical device number 5 is free, and they use this number in all device operations.

The piece of information needed by almost every subroutine is the address of a file control block. This block is built in memory when the file is OPEN'ed (or created), and contains all sorts of pertinent information about the file. It should not be touched (even accidentally, if you can manage that) except by the file subroutines. If it does get wiped out or screwed up, your file will almost certainly go the same route. If you are extremely unlucky, you may even muddle some other file that doesn't even belong to you.

SUBROUTINE CHKSUM

Name: CHKSUM

Function: to both check and recompute the checksum in a file control block.

Parameters:
GR1: File control block address
GR14: Return address
GR15: Address of CHKSUM
Note: no save area required

Returns: A return code will be in GR15

0 indicates checksum in FCB was correct

4 indicates it was not, but CHKSUM has recomputed it, and the FCB now has a correct checksum.

Description: You should always call CHKSUM before and after modifying the FCB. But very few routines do, so don't worry about incorrect checksums. It only gets a writeup because it is there, and people might wonder about it.

SUBROUTINE CLOSE

Name: CLOSE

Function: To write out any modified data contained in buffers assigned to the file.

Calling sequence:

GR1:	File control block address.
GR13:	Pointer to a scratch area of 300 bytes.
GR14:	Return address
GR15:	Address of CLOSE.

Returns: GR15 has a return code

Normal return code is 0.

Return code of 4 indicate the checksum in the FCB was incorrect. Don't worry about it. Nobody maintains the checksum anyhow. The file will still be (correctly) closed.

A return code of 8 indicates some serious error occurred. Likely causes: an unrecoverable disk error, badly clobbered FCB, or bad FCB address.

Description: CLOSE looks through the buffers assigned to the file, writing out any that need writing, and turning off the must-be-written bits for those buffers.

It is possible to read or write a previously CLOSE'd file, but it should then be CLOSE'd again when you are really through with it.

SUBROUTINE CREATE

Name: CREATE

Function: To 1) Make an entry in the Volume Table of Contents on a specified disk pack for a file by the specified name.

2) Make an initial space allocation for that file on the same disk pack.

3) OPEN the file so that it may be written or read.

Parameters:
GR1: Pointer to a parameter list.
GR13: Pointer to a working area of 475 bytes
GR14: Return location
GR15: Address of CREATE

Parameter list:

BYTES 0-15:	File name, blank-filled, left-justified
BYTES 16-19:	FCB link or 0
BYTES 20-23:	Address of 3520 byte buffer
BYTES 24-25:	Length of that buffer
BYTES 26-31:	Volume name of disk pack on which to place this file. If this parameter is all (hex) zeroes, the file will be put on any available system volume.
BYTE 32:	Number of tracks to initially allocate.
BYTE 33...	Used to create the FCB.

Note that up to BYTE 25 the parameter list is the same as that for OPEN.

Returns: On return, a return code is left in GR15 as follows:

0 means AOK.

4 means the file already exists (CREATE does not open the file in this case - it just punts)

8 means there is no space available (either on the disk pack you asked for, or anywhere if you were indifferent)

12 means there was some serious error. Hardware or software.

Description: CREATE just looks to make sure the file does not already exists, and if it does not, it creates a VTOC entry for that

file. It then allocates the requested number (up to 100) of tracks (not records) to the file in a single extent. The file is then initialized and opened as through it were going to be a line file.

FILE OPERATION SUBROUTINES

SUBROUTINE DESTRY

Name: DESTRY

Function: To de-allocate all disk areas previously allocated to a file, and destroy the Volume Table of Contents (VTOC) entry for the file itself.

Calling sequence:
GR1: pointer to a 16-byte file name
GR13: pointer to a scratch area (575 bytes)
GR14: address of return location
GR15: address of first location in DESTRY

Returns: GR15 has return code

Normal return code is 0, indicates successful completion.

Return code of 4 indicates the file couldn't be found to be destroyed. Sorry about that.

Return code of 8 indicates something else went wrong. Probably something wrong with the subroutine, but then it might be in the call such as not providing enough scratch area, etc.

Description: DESTRY first of all locates the catalog entry for the named file, and assuming it finds that, releases all extents allocated to the file. It then marks the VTOC entry itself as available for a new entry.

SUBROUTINE GETDSK

Name: GETDSK

Function: To acquire some space for a specified file, and record the additional allocation in the VTOC entry for that file.

Parameters:
GR1: Address of a parameter list
GR13: Address of a 400 byte working area
GR14: Return address
GR15: Address of GETDSK

Parameter list format:
BYTES 0-3: FCB address
BYTES 4-7: Where to put extent address
BYTES 8 : Number of tracks (not records) requested

Returns: A return code is left in GR15 as follows:

0 indicates AOK. An extent address has been placed in the address given in tne parameter list, and looks as follows:

BYTES 0-5: Disk name where it is
BYTES 6-9: CCTT of beginning of the extent.

4 indicates there is not enough space available, on the volume on which this file resides, to satisfy the request. The parameter list is modified (BYTE 8) to indicate the largest extent (in tracks) available on this volume. (It may be 0)

8 indicates the usual serious snark that should never happen.

Description: GETDSK searches the chain of available disk space, trying to find a single extent to fill the request. If none is available, it determines what the largest extent available is, and modifies the parameter list to indicate that.

If an extent is available, the smallest one which is large enough is allocated to the file, and the VTOC is updated in indicate that the allocation has been made.

SUBROUTINE OPEN

Name: OPEN

Function: To search for a specified file, (if it exists) set up a file control block for it, and initialize the buffers for subsequent READ or WRITE operations.

Calling sequence: Pointer to a parameter list (and future file control block area).
GR13: Pointer to a working area of 350 bytes.
GR14: Return address
GR15: Address of OPEN.

Returns: GR15 has a return code as follows:

0 is the normal return.

4 indicates that the named file could not be found anywhere. (This generally means it isn't anywhere.)

8 indicates something serious (like an unrecoverable disk error or an error in the calling sequence) has occurred.

Parameter list:

BYTES 0-15: Name of the desired file. Left-justified, blank-filled

BYTES 16-19: Address of next FCB link or 0.

BYTES 20-23: Address of a buffer to be used for reading and writing the file.

BYTES 24-25: Length of this buffer (in bytes)

Description: If the file named in the parameter list can be found, the parameter list (both the data and the memory where the list is) is used to create a File Control Block for the file.

This supplied buffer is then used to read in the first record of the first extent assigned to the file. If this looks like a line (system) file, it is checked to see if this file is extended, and in this case an additional buffer may be acquired and added to the FCB.

It should be emphasized that the parameter list should be contained in an area at least 80 bytes long, and that this area will have become an FCB if the return code is normal.

SUBROUTINE READ

Name: READ

Function: To read a specified line from a specified file into a specified area (now you even know what the parameters are).

Parameters:
GR1: Pointer to a parameter list
GR13: Address of a working area of 400 bytes
GR14: Return address
GR15: Address of READ

Parameter list format:
BYTES 0- 3: Number of desired line (32-bit signed integer)
BYTES 4- 7: Where to put the line
BYTES 8-11: Address of file control block

Return code: A return code is always left in GR15 as follows:

0 indicates all went well

4 indicates line does not exist. The entry (see below) for the following line is returned.

8 indicates line is beyond any in file

12 is some serious error indication.

For returns of 0 and 4, GR1 contains additional information. For an RC of 0, GR1 indicates the number of bytes in the line (guaranteed to be < 256). For a RC of 4, the number of the following line is returned in GR1.

Description: READ attempts to locate a line with the number given. If one does exist, it moves it to the specified area. If there is no line in the file with that number, it determines if there is no line with a number that high, or locates the line following the (non-existent) one asked for, and makes the appropriate indication on return.

SUBROUTINE READL

Name: READL

Function: To determine the line number of the last number in the file and return it.

Parameters:
GR1: Address of file control block
GR13: Address of a 400 byte working area
GR14: Return address
GR15: Address of READL

Returns: Return is always made with a return code in GR15.

0 indicates o.k. return, and in this case, the line number of the last line in the file will be in GR1

4 indicates some kind of a nasty error.

Description: READL searches the file for the last line in it, determines the number of that line and returns it. In the case of an empty file (no last line) a line number of 0 is returned.

SUBROUTINE READS

Name: READS

Function: To read sequential lines from a file (i.e., like 'READ', except the line number is automatically supplied)

Parameters:
GR1: Address of a parameter list
GR13: Address of a 400 byte working area
GR14: Return address
GR15: Address of 'READS'

Parameter List Format:

BYTES 0-3: Number of first line to be read on first call, number of last line read on subsequent calls.

BYTES 4-7: Where to put requested lines.

BYTES 8-11: File control block address.

BYTES 12-15: Pointer to a fixed full-word area which will be zero on the first call to READS. Thereafter READS will use it to store information it must have from call to call (so the same location must be given with every call, and its contents must not be changed between calls).

Returns: A return code is always left in GR15 as follows:

0 indicates everything went fine

4 indicates an end-of-file condition

8 indicates a bad error.

If the return code is 0, R1 contains the byte count for the line read.

Description: READS is used when one wants to sequentially access each line in a file, without worrying about what the actual line numbers. The first call on READS must specify what line to start at, and thereafter, each subsequent call will return the next line in the file (until the end is reached.)

SUBROUTINE RELDSK

Name: RELDSK

Function: To return to free storage some portion of the disk storage allocated to a file.

Parameters:
GR1: Pointer to a parameter list.
GR13: Pointer to a 400 byte working area
GR14: Return address
GR15: Address of RELDSK

Parameter list:
BYTES 0-3: Pointer to file name.
BYTES 4-7: Pointer to a 14 byte extent descriptor.

The extent descriptor has the following form:

BYTES 0- 5: Volume name (6 bytes)
BYTES 6- 9: Address of beginning of the extent as CCTT.
BYTES 10-13: Address of last track in the extent as CCTT.

Returns: A return code is always left in GR15, as follow:

0 indicates a normal return

4 indicates file VTOC entry could not be found

8 indicates extent to be released isn't even allocated to the file

12 indicates something even worse. (hard to say what)

Description: RELDSK very simply removes the specified extent for the VTOC entry for that file and returns it to the list of available extents. It's returning it to the list that is so messy, and this is the module which is the only section of the file routines <u>not</u> compatible with OS/360 file formats. (It does not order the TYPE 5 record entries, for those of you who read system programmer's guides)

SUBROUTINE SCRTCH

Name: SCRTCH

Function: To erase all the lines (records) in a file without releasing the disk area assigned to the file or destroying the file itself.

Calling sequence:

GR1: File control block address.
GR13: Address of a working area of 350 bytes.
GR14: Return address.
GR15: Address of SCRTCH

Returns: GR15 contains a return code as follows:

0 is the normal (like OK) return.

4 indicates an error of some seriousness. Probably GR1 didn't <u>really</u> contain an FCB address

Description: SCRTCH essentially removes all the lines from a file, but keeps all the disk storage allocated to it. Thus you have an empty file, which is why the command that calls this subroutine is called $EMPTY.

SUBROUTINE WRITE

Name: WRITE

Function: To add, replace, or delete a line in a specified file.

Calling sequence:
GR1: Pointer to a parameter list.
GR13: Pointer to a working area (of 475 bytes)
GR14: Address of return location
GR15: Address of WRITE

Parameter list format:
BYTES 0- 3: File control block address.
BYTES 4- 7: Number of line to be written.
BYTES 8-11: Address of line to be written.
BYTE 12 : Length of line to be written.

Returns: A returns code will be in GR15 on return as follows:

0 is the normal return

4 an error return. This almost always means the file got too big for the disk pack or the software.

Description: The worst case path through WRITE is long and tortuous, but the average path is quite short and simple. WRITE always requires a line number as an argument. If a line with that number already exists in the file it is replaced by the one given to WRITE. If there is no line by that number, the supplied one is added to the file (at the place specified by the line number).

If the line is 0 bytes long, it is deleted (or the call is ignored, if there is no line by the number given in the call).

WRITE takes care of allocating more space to the file, as needed, up to a limit of 218 records. This allows a file to be about 800,000 characters long (about 10,000 card images). In figuring maximum file size one must remember that every line incurs an overhead of 8 bytes (see writeup on file formats), in addition to its actual length in characters.

FILE SUBROUTINES INTERNAL STRUCTURE

In addition to the subroutines called by the MTS supervisor, the file system contains other subroutines (hereafter called 'internal' subroutines), which are called by the subroutines called by the MTS supervisor (hereafter called 'external' subroutines).

The attempt here is not to carefully define the function and calling sequence for each of these subroutines, but rather to give the reader an idea of the structure of the file system subroutines in general.

First, a quick listing of the subroutines and their function.

DSKOP: Disk operation subroutine executes an error checked disk operation on a specific disk.

VOLGET: Locates a disk pack by its 6-character volume name, and acquires it as logical device 5.

VTOCS: Searches for a file, by name, on the disk packs assigned to MTS

VOLREL: Allows caller to modify the list of disk packs assigned to MTS, by volume name.

READT: Read in a single, specified track of a disk pack, after locating an available spot in the file control block (FCB) to record the reading.

READR: Like 'READT', only it assumes an available buffer entry in the FCB is known.

WRITET: Writes a specified track, in some FCB buffer entry, onto the given disk pack address.

GTZ: Locates a free record (name stands for 'get type 0') in the VTOC of a file, returns its address to caller. Also updates information indicating that record is no longer free.

FLINE: Searches line directories for a specified line. Returns directions for finding the requested line, the number of the next line, or an end-of-file indication (it depends...).

Before discussing the flow of control in the internal routines, there is one other table which should be described, and that is the one called 'DSKTAB', which indicates what disk packs (by name) are available to MTS, what physical device (if known) they are on, and their status.

Each DSKTAB entry is 124 bytes long (which leaves lots of room for status), there is one for every disk pack available to MTS, and the format of each entry is as follow:

BYTES 0-17	Used by disk unit check routines as working area, in case of disk errors
BYTES 18-23	Volume name for this entry (6 EBCDIC characters). If this area is hex 'FF's, this entry is ignored.
BYTES 24-27	Name of device on which this volume is mounted (4 EBCDIC characters). If this area is hex 'FF's, it is assumed that it is not yet known where this device is.
BYTES 28-123	Data portion of TYPE 4 record for this volume (see 'System Programmer's Guide' for format.)

The table is terminated by 4 bytes of hex FF's in what would be the first 4 bytes of the next entry.

The only three subroutines concerned with DSKTAB are VOLGET, VOLREL, and DSKOP.

VOLREL may be called to modify the disk table entries (either adding or deleting names) and assign a new modified set of disk packs to MTS while it is running.

VOLGET is called to acquire a disk pack (really a physical device) from the system. Under some circumstances this may require a search of all available physical devices to see which one the desired volume is mounted on. Normally, of course, it just requires a search of DSKTAB, and a call to the supervisor GTUNIT routine.

All disk operations performed by the file system except VOLGET are done through the subroutine DSKOP. DSKOP requires that the user already have called VOLGET to acquire the appropriate volume, and then requests an error checked execution of a specified CCW list. This CCW list must meet the specifications given for error checking in the disk unit check routines, and in addition must meet the following restriction:

1. The first command in the CCW list must be an unchained seek.
2. There must be one or more chained CCW's following the first, none of which may be seeks.

DSKOP does not check CCW lists for validity of command or data in any way.

The subroutines GTZ and VTOCS are concerned primarily with maintenance of the volume table of content (VTOC) of the disk packs assigned to the system.

VTOCS, when called, searches the VTOC's of all disk packs assigned to the system (via DSKTAB), for an entry with a specified name. Though OS names can be up to 44 bytes long, MTS allows (and looks for) only 16 byte names.

GTZ is used to find a 'free' record in the VTOC of a specified volume. GTZ will find the first free record (if there is one) in the VTOC, and return it's address to the caller. If the VTOC is extended in this process, the TYPE 4 record is appropriately updated.

NOTE

The general reference for the format of VTOC's records in them, etc, is System Control Blocks, Form C28-6628.

The rest of the internal subroutines are concerned with maintenance of the files themselves rather than the VTOC.

READT and WRITET read and write the records of a file. They may be called from several of the external subroutines and WRITET may, in fact, be called by READT to write out a track to free a buffer in the FCB. READT, when called, searches the buffer entries to find

1. A buffer which is not in use
2. A buffer which is of lower priority than the one in use (i.e., the record now in the buffer is of lower priority)
3. A buffer which does not need to be written.

That is to say, READT attempts to use the least important area to read the requested track into. It is also possible to call READT with a conditional request, it will only read the track if there is a free buffer, or one with a lower priority.

FLINE searches line directories for a specified line. It is called by READ, READS, READL and WRITE. It first searches for the record of the line directory (if there is more than 1) in which the entry for the desired line is to be found, if it is there, and then uses a half-interval search on that record to locate the entry. The only subroutine FLINE calls is READT, and it can indicate several possible things on return:

1) Desired entry found (it tells you where the line itself is).

2) No such entry (it tells you what the number of the next line is).

3) End-of-file (the line you requested has a higher number than any in the file).

The actual calling sequences for each of the internal subroutines is delineated only in the assembly listings themselves, which should also be considered the final arbitrator in how the subroutines themselves work.

Indexed Words and Phrases

Indexed Words and Phrases

Indexed Words and Phrases

Indexed Words and Phrases

Indexed Words and Phrases

Indexed Words and Phrases