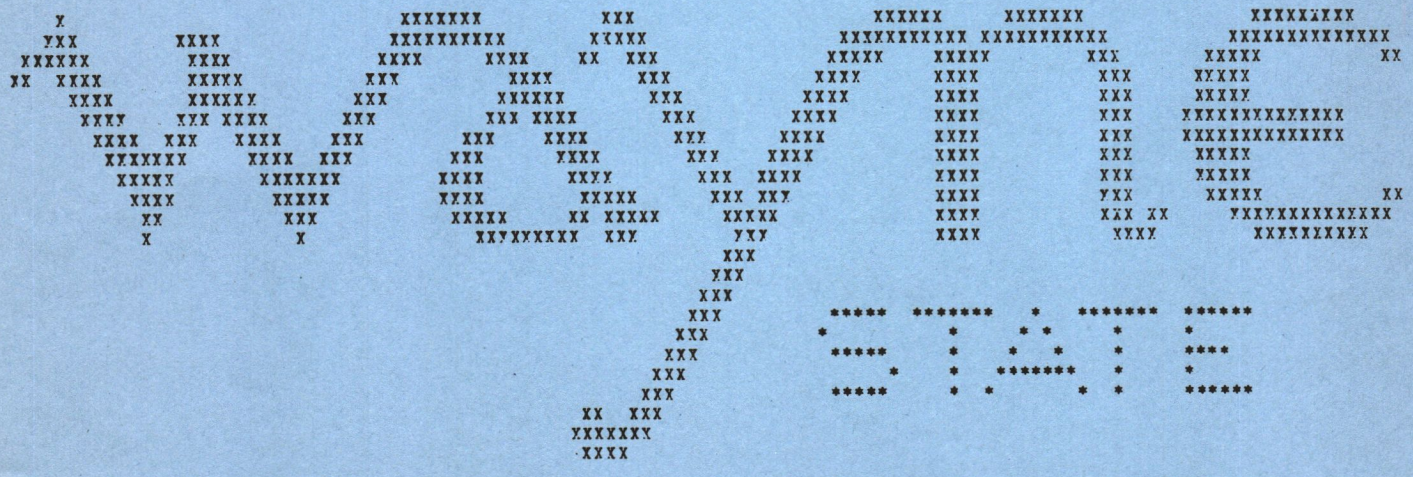
VOLUME 1

THE SYSTEM

MTS



APRIL 1974

M T S

The Michigan Terminal System

VOLUME 1: MTS -- THE SYSTEM

Third Edition

April 1974

Revised

Wayne State University
Computing and Data Processing Center
Detroit, Michigan 48202

****
*
* This obsoletes the April 1971 printing. *
*
****

DISCLAIMER

This manual represents an attempt to document the "current" state of the Michigan Terminal System (MTS) as used at Wayne State University. It is a revision of MTS and the Computing Center (Volume 1, January 1973) published by the University of Michigan Computing Center at Ann Arbor and MTS -- The System (Volume 1, April 1971) published by the Wayne State University Computing and Data Processing Center.

As the system is developed, some sections of this manual will become obsolete. The user should refer to the file HELP:NEWS and the WSU CDPC newsletter "Overflow" for the latest information about modifications to MTS.

April 1974

U OF M PREFACE TO THE THIRD EDITION

The software developed by the University of Michigan Computing Center for the operation of the 360/67 dual-processor computer can be described as a multi-processor supervisor which handles a number of resident, re-entrant programs. Among these is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file handling and accounting maintenance. The majority of users interact with the computer's resources by means of the latter operating subsystem. The various MTS manuals are concerned principally with describing the use of this subsystem.

This, the Third Edition of the MTS manual, is a major revision and reorganization of the Second Edition (December 1, 1967). The third edition will consist of a series of 10 or more separate, self-contained volumes, to be published over a period of many months. Eight volumes in the third edition have already appeared.

MTS and the Computing Center, Volume 1, revised January 1973
Public File Descriptions, Volume 2, revised April 1971
Subroutine and Macro Descriptions, Volume 3, June 1970
Terminals and Tapes, Volume 4, September 1972
System Services, Volume 5, December 1971
SNOBOL4, Volume 9, March 1969 (currently out of print)
BASIC in MTS, Volume 10, August 1971
Plot Description System, Volume 11, April 1971

Others are in preparation. The numerical order of the volumes is not necessarily the chronological order; for example, Volumes 2, 3, and 9 appeared before Volume 1. However, in general, the higher the number, the more specialized the volume is. Volume 1, for example, introduces the user to MTS and the Computing Center, while Volume 10 deals exclusively with BASIC. The attempt to make each volume mostly self-contained naturally results in a certain amount of repetition. Such things as public file descriptions, for example, may appear in more than one volume. However, with this arrangement, the user is not forced to buy the entire set of manuals, but only those that serve his immediate needs. And, hopefully, each volume will be smaller and of a more manageable size than before.

Lynn R. Leader
Richard A. Salisbury
Mary Ann Wilkes
General Editors

April 1974

WSU PREFACE TO VOLUME 1

The April 1974 revision of Volume 1 reflects the changes that have been made to MTS at WSU since the first printing of Volume 1 in April 1971. It also incorporates relevant parts of MTS AND THE COMPUTING CENTER (Volume 1) published by the University Michigan in January 1973. The changes made include deletion of material specific to the University of Michigan, alterations necessary to reflect the differences between the two systems, and additions to reflect the current development at WSU.

We are extremely grateful to the University of Michigan Computing Center for permission to publish this revision of their manual. The following paragraph is quoted from their preface to Volume 1 of February 1971:

"While we would like to acknowledge the contributions of many people to this volume, it is the sad truth that at this point the editors (and others) have rewritten and restructured each others writeups and other people's writeups over and over to the extent that it is impossible to assign the exclusive blame for any part of this manual to anybody."

We would also like to acknowledge that much of the raw fodder for this manual also came from the University of Alberta Computing Center and the University of British Columbia Computing Center. It is difficult at this time to even identify from which sources parts of this manual came. More people have contributed to this manual then could possibly be identified; to each we are grateful.

The writeups in this volume refer occasionally to other volumes that have not been published. While this is regrettable, we felt that it would be better to include the reference in the hope that the the volume would be published before the next printing.

The last page of this manual is an Update Request Form which must be returned to the WSU Documentation Librarian to insure that future updates are received. It can also be used for suggestions or corrections.

This publication obsoletes WSU Technical Memos 1 and 12 (TM001-000 and TM012-000).

Richard L. Wiersma
Russell W. Pratt
Barbara B. Wolfe
Revision Editors

April 1974

Contents

April 1974

DOCUMENTATION

Information on computer utilization at WSU is obtained primarily from documentation. The documentation sources may be many: WSU Computing and Data Processing Center (CDPC) publications, U of M Computing Center publications, IBM, and manuals and texts from other sources. This chapter describes the publications necessary for utilization of the operating system MTS (Michigan Terminal System) at WSU and references other documents for some of the more commonly used computer applications.

The Academic Services Department of the Computing and Data Processing Center attempts to compile and disseminate complete, accurate, detailed, and up-to-date information about the Center's equipment, operating systems, and administrative policies for a wide spectrum of users. The principal publications are the MTS manuals which often rely heavily on U of M's MTS manuals. U of M's series of MTS manuals when completed will consist of about a dozen 300-page volumes describing MTS in detail. The WSU CDPC will make completed U of M manuals available on campus when they likewise apply to MTS at WSU. When necessary, they will be revised and republished here to reflect changes made to MTS at WSU. Currently volumes 1 through 4 and 12 have been revised and republished for use at WSU while volumes 5, 10, and 11 are U of M publications.

Other introductory publications besides MTS manuals are also recommended at WSU. These publications are more specific, less costly to produce, and can thus be made available to students and other users at a lower cost than the more voluminous MTS manuals. An informal newsletter called OVERFLOW is published as needed (at least once per month) to inform users of changes taking place at the WSU CDPC. Other documents called Technical Memos are published by the Academic Services staff to provide detailed accounts of changes and additions to MTS or to provide tutorial information in some aspect of a computer application. Some of the Technical Memos are subsequently incorporated into MTS volumes. U of M Computing Center also produces CCMemos much for the same purpose. Often these CCMemos are equally applicable at WSU and those which apply are made available to users here. Other WSU application program writeups exist which are in the same category as Technical Memos but were not labeled as Technical Memos when they were published.

The WSU CDPC tries to recover the printing costs of some documents it produces. These are sold through the main branch of the WSU bookstore. Other documents are freely given to users. Documents which the CDPC must purchase elsewhere are sold to users at our cost; these are usually either low quantity items or from lesser known publishers. Still others which come from large volume publishers and are high volume items are best purchased through the WSU bookstore.

Sources for documentation are indeed complicated. However, the Academic Services Documentation Librarian can be consulted for details. General guidelines are as follows. MTS manuals are purchased from the WSU bookstore (both those published by WSU and U of M). Updates to MTS manuals are sent free of charge to any user who returns the Update Request Form (the last page of the manual). OVERFLOW is mailed free of charge to any user on the mailing list; contact the editor to be added to this list. Project directors are automatically included on the OVERFLOW mailing list. Some WSU Technical Memos are distributed free of charge upon request to the Documentation Librarian; other high volume Technical Memos are sold at the WSU bookstore. Application program writeups are generally distributed free of charge for a single copy by the Documentation Librarian. U of M CCMemos are sent to users by the Documentation Librarian free of charge upon request. IBM manuals must be paid for by the user. The WSU bookstore can place orders for IBM manuals. If the user is a WSU faculty or staff member and if he wants IBM's updating service for IBM manuals, he may request IBM manuals from the Documentation Librarian. The CDPC will then bill him for the cost of the IBM manuals and any subsequent replacement manuals issued by IBM. Users external to the University who do not have access to the WSU bookstore can request any documents from the Documentation Librarian and be billed for these items by the CDPC.

REFERENCE CENTERS

Two locations on campus are maintained by the Academic Services Documentation Librarian as reference centers for documentation on computer utilization. These centers contain all commonly used documentation at WSU and also major manuals for use of the University of Michigan Computing Center and the Michigan State University Computing Center through the MERIT Computer Network. (See MTS Volume 4, "The MERIT User's Guide".) These locations are the main circulation desk of the WSU Science Library and room 333 Administrative Services Building 2, 5950 Cass Avenue. Documentation may be perused at these locations which may be helpful in determining if purchase is necessary.

WSU MTS DOCUMENTATION

The following list describes the WSU applicable volumes of the third edition of the MTS manuals:

Vol. 1, MTS -- The System, revised April 1974, 480 pp.
Introduces MTS and describes the command language, files and devices, general batch and terminal use.

Vol. 2, Public File Descriptions, revised August 1971, 316 pp.
Describes each of the public files available in MTS.

Vol. 3, Subroutine and Macro Descriptions, revised January 1972, 434 pp.
Describes the subroutines and subroutine libraries, and macros and macro libraries that are available in MTS.

Vol. 4, Terminals and Tapes, revised September 1973, 142 pp.
Contains sections on conversational use of MTS, the MERIT network, and magnetic tapes.

Vol. 5, System Services, December 1971, 232 pp.
Describes the SORT utility program, text processing programs, the dynamic loader, and IOH/360.

Vol. 10, BASIC in MTS, August 1971, 336 pp.
Describes the BASIC language as implemented for MTS.

Vol. 11, Plot Description System, April 1971, 164 pp.
Describes the use of the CALCOMP plotter on MTS.

Vol. 12, Pitt Interpretive Language, revised February 1974, 110 pp.
Describes PIL, an interactive language, on MTS.

WSU INTRODUCTORY PUBLICATIONS

Facilities & Services of WSU Computing and Data Processing Center for Academic Users, February 1973, 42 pp.
Describes the facilities and services that are available to users of the Wayne State University Computing and Data Processing Center.

Introduction to MTS @ WSU, January 1974, 82 pp.
Describes how to use the MTS command language at WSU to those users who are unfamiliar with the system.

Abstracts of Available Software, issued quarterly, 54 pp.
A listing of abstracts of many programs available at WSU.

OTHER DOCUMENTATION RELEVANT TO MTS

MTS Reference Card, WSU CDPC, February 1972.
This concise, pocket-size card contains the MTS command language, editor commands, debug commands, CDPC telephone numbers, FORMAT (*FMT) commands, MTS BASIC, FORTRAN IV and other frequently used items.

CONTOUR, Richard Wiersma, WSU CDPC, March 1973, 50 pp.
CONTOUR, a generalized surface approximation, contouring, and plotting programs is described.

Introduction to SNOBOL4, ed. E. J. Fronczak, U of M Computing Center, February 1970, 25 pp.

The SNOBOL4 Programming Language, Griswald, Poage and Polonsky, Prentice-Hall, rev. ed., 1971.
This programming language, developed at Bell Telephone Labs, is designed to be used chiefly for manipulation of character strings.

SPITBOL Users' Guide, ed. Ken DeJong, University of Michigan Computing Center and Richard Wiersma, Wayne State University Computing and Data Processing Center, WSU CDPC, September 1973, 62 pp.
This is a fast version of SNOBOL4 developed at the Illinois Institute of Technology.

APL/360, An Interactive Approach, Gillman and Rose, Wiley, 1970.

APL/360 Reference Manual, Pakin S., Science Research Associates, 1972.

Digital Computing, FORTRAN IV, WATFIV, and MTS (with *FTN and *WATFIV), Carnahan and Wilkes, published by the authors, Chemical Engineering Department, University of Michigan, 1972.

A GPSS Primer, Schreiber, preliminary printing, published by the author, School of Business Administration, University of Michigan, 1972.

A Compiler Generator, McKeeman, Hornung, and Wortman, Prentice-Hall, 1970.
Describes the XPL programming language.

Merit Computer Network User's Guide, ed. Susan E. Colman, preliminary printing, published by Merit Computer Network, U of M, MSU, WSU, April 1973, 60 pp.

SIMSCRIPT II.5 Programming Language, Kiviat, Villanueva, and Markowitz, Consolidated Analysis Centers Inc., 1973.

SIMSCRIPT II.5 Reference Handbook, Consolidated Analysis Centers Inc., 1972.

SIMSCRIPT II.5 Programmer's Manual, WSU CDPC, May 1972, 56 pp.

The above three manuals comprise the complete documentation for the SIMSCRIPT II programming language at WSU.

Many other books and manuals could be listed in this section. Additional references are in Abstracts of Available Software and the MTS file HELP:DIRECTORY.

April 1974

IBM MANUALS

The MTS manuals will eventually describe fully the languages that were developed for use in MTS. They will also contain full descriptions of languages developed elsewhere and adapted to use in MTS, such as PIL, but for which descriptions are not readily found in the existing literature. However, languages developed by IBM and adapted to MTS are described only briefly in the MTS manuals. For these languages, the user should refer to the appropriate IBM manuals for a complete description.

The IBM manuals listed below are important to a better understanding of use of the computer and the specific IBM language processors available in MTS. These manuals can be purchased from the WSU main bookstore, but users must supply the correct form number when ordering them. The CDPC Academic Services Documentation Librarian has a complete bibliography of IBM publications, and will gladly assist users in finding the correct form number.

Hardware Description
- IBM System/360 Principles of Operation--form GA22-6821
- IBM System/360 Model 67 Functional Characteristics--form GA27-2719
- IBM System/360 System Summary--form GA22-6810

Assembler
- IBM System/360 Assembler Language--form GC28-6514

Reference Data
- IBM System/360 Reference Data--form GX20-1703 (also known as the "green card")

FORTRAN
- IBM System/360 FORTRAN IV Language--form GC28-6515
- IBM System/360 FORTRAN IV Library Subprograms--form GC28-6596
- IBM System/360 FORTRAN IV Library: Mathematical and Service Subprograms--form GC28-6818
- IBM System/360 Scientific Subroutine Package--form GH28-0205

PL/I
- IBM System/360 PL/I Reference Manual--form GC28-8201
- IBM System/360 PL/I Subroutine Library--form GC28-6590
- IBM System/360 PL/I (F) Programmer's Guide--form GC28-6594 (Although IBM system-dependent, this manual also contains much useful information on writing correct PL/I programs.)
- A PL/I Primer--form SC28-6808
- A Guide to PL/I for FORTRAN Users--form SC20-1637

FORTRAN to PL/I Converter
- System/360 Conversion Aids: FORTRAN IV-to-PL/I Language Conversion Program for IBM System/360 Operating System--360C-CV-710--form GC33-2002

APL

APL/360 Primer--form GH20-0689
APL/360 User's Manual--form GH20-0683

ALGOL

IBM System/360 Operating System ALGOL Language--form GC28-6615
IBM System/360 Operating System ALGOL Programmers Guide--form GC33-4000

COBOL

IBM System/360 American National Standard COBOL--form CG28-6396 (Although IBM-system-dependent, this manual also contains much useful information on writing correct programs.)
IBM System/360 American National Standard COBOL--form GC28-6399

CSMP

System/360 CSMP: Application Description--form GH20-0240
System/360 CSMP: User's Manual--form GH20-0367
System/360 CSMP: System Manual--form GY20-0111

GPSS

GPSS/360 Introductory User's Manual--form GH20-0304
GPSS/360 User's Manual--form GH20-0326

MPS

Mathematical Programming System/360: MPS Application Description--form GH20-0136
Mathematical Programming System/360: MPS Control Language User's Manual--form GH20-0290
Mathematical Programming System/360: MPS Linear and Separable Proramming User's Manual--form GH20-0476
Mathematical Programming System/360: MPS Messages Manual--form GH20-0603

April 1974

INTRODUCTION TO MTS

The IBM 360/67 can serve a large number of users concurrently, offering each a wide variety of services. The job of keeping track of all the programs in the machine and of devoting some attention to each of them every second or two is handled by the MTS operating system (Michigan Terminal System). In order to request service from the computer, you must first identify yourself to MTS and then communicate your requests. This communication is done through the MTS Command Language. Because the system is both powerful and complex, the command language of MTS is a rich one. Fortunately, MTS usually has a default option wherever it offers you a choice; that is, if you don't state your choice explicitly, a plausible assumption is made and the job continues. In the following sections, some of the more important commands are discussed. For a complete listing, see the section entitled "A Brief Overview of MTS" in this volume.

You may operate within MTS in one of two modes - batch or conversational. In conversational mode, you sit at a terminal and communicate directly with MTS through a typewriter-like keyboard. MTS processes each command as it is received and reports the results. Based on the results, you can decide what command to give next. Thus, conversational mode of operation is highly interactive.

Batch mode, on the other hand, is not interactive. You must completely pre-plan all requests, punch them on cards, and submit them to the Computing and Data Processing Center (CDPC), directly or through a distribution station. Some time later -- minutes or hours, depending on how long your program is and how busy the computer is -- you may pick up the results. Feedback is not immediate. Details on submitting a program using a terminal or in batch are found in the manual Introduction to MTS @ WSU (January, 1974) and the sections "Conversational Usage of MTS" and "Batch Usage of MTS" in this volume.

The command language and its usage are essentially the same for both batch and conversational modes. The primary difference is that in conversational mode MTS replies after most commands, confirming what it just did or prompting you for further information. The descriptions that follow apply to both modes; any substantial differences between them are noted as they occur.

INTRODUCING YOURSELF TO MTS

Here, and in later sections, the emphasis is on illustrative examples accompanied by "rules". This is not the whole MTS story by any means, but it should serve as an introduction to the more detailed descriptions in the rest of the manual.

Example 1 is a trivial job that illustrates four rules about the MTS command language:

```
$SIGNON MYID PW=SESAME 'W NAME'
$SIGNOFF
```

Example 1.

Rule 1: The first character of a command to MTS is a "$". In batch mode, the "first character" is the character in column 1 of a card. In conversational mode, the "first character" refers to the first character you type in a line. (A blank is considered to be a character.) Though there are cases in which the "$" need not appear, it is always acceptable to include it.

Rule 2: The first card or typed line must say $SIGNON and must be immediately followed by one or more blanks and your user identification code. This first part of the identification is the four-character ID code (e.g., MYID) given to you by the Computing Center. The next part is the 1-6 character password associated with the ID. More is given on passwords below.

Rule 3: The last command, $SIGNOFF, tells MTS that you have finished using its facilities. MTS then gives statistics on the completed run, e.g., how much it cost, how long it took, etc. The $SIGNOFF command is not necessary to terminate a batch job.

Rule 4: For batch users only, a delivery code and name should appear within single quotes. The delivery code indicates where the printed and/or punched output is to be delivered. The letter "W" indicates that the output is to be sent to room 73 of the Science Library. A blank indicates that the output should be retained at the control desk of the CDPC, 5925 Woodward Avenue. The name is used for identification and should be separated from the delivery code by a blank or comma. For a listing of delivery codes, see Facilities and Services of MSU Computing and Data Processing Center for Academic Users or the file HELP:DELIVERY.

The purpose of the password is to keep others from using your signon ID. It is prudent to conceal it. One step toward this goal is to leave the password off the $SIGNON command and to supply it on the next card or line. This second card is not printed with your output in batch mode, and it is recommended that you turn off the printing on the keypunch or terminal when you type it. The "three-line" sequence in Example 2 has exactly the same effect as Example 1 and is the preferred method of signing on. This method is used in all subsequent examples.

```
$SIGNON MYID 'W NAME'
SESAME
$SIGNOFF
```

Example 2.

In conversational mode, if you don't include the password on the first line, MTS prompts you for it by typing "ENTER USER PASSWORD".

When you get a signon ID from the CDPC, a password is already assigned to it. You probably will want to change it to something more meaningful and hence more easily remembered. The $SET command is used to change your password. To change from your current password, OLDPAS, to a new one, NEWPAS, you can include the $SET command as part of any job:

```
$SIGNON MYID '  NAME'
OLDPAS
$SET PW=NEWPAS
$SIGNOFF
```

Example 3.

From this point on, your password is NEWPAS.

When you $SIGNON in batch, MTS allows you a maximum of 30 seconds of processing time, 50 pages of printed output, and 0 punched cards. If you want more or less of any of these items, you can specify the request on the $SIGNON card as shown in Example 4 below.

```
$SIGNON MYID T=90 P=10 C=50 '  ME'
```

Example 4.

Rule 5: To set the time, you write "T=time" in seconds. This example requests 90 seconds. If you prefer to use units of minutes, you may write T=1.5M, where M indicates minutes. To specify pages and cards, you write "P=number of pages" and "C=number of cards". This example asks for 10 pages and 50 punched cards.

These parameters serve only as a maximum; you may use less but you cannot use more. For a more detailed description of these and additional options for the $SIGNON command, see the description of the command language in this volume. In terminal mode, these options are ignored.

NOW THAT MTS KNOWS WHO YOU ARE

Now you can get the computer to do some real work for you. Let's say that you have some problem for which you have written a FORTRAN or WATFOR program. In order to run this program, i.e., to have the computer carry out your instructions, several steps are required:

1. The program you write in FORTRAN or WATFOR or any other language is known as the source program. A translator must translate your source program to a machine language object program. If you wrote the program in FORTRAN, then you use the FORTRAN compiler to translate your FORTRAN statements. If you used WATFOR statements, the WATFOR compiler must do the translating. The same is true for other languages.

2. The object program must be loaded into computer memory.

3. MTS can then execute your object program, that is, it can carry out the instructions in your program. Let's look at some sample WATFOR jobs.

```
$SIGNON MYID '  NAME'
SESAME
$RUN *WATFOR
$COMPILE
      READ,X
      U=SQRT(X)
      PRINT, X,U
      STOP
      END
$DATA
21.67035
$STOP
$SIGNOFF
```

Example 5.

Rule 6: $RUN *WATFOR tells MTS to execute the program contained in the public file *WATFOR. The program in file *WATFOR is the WATFOR compiler. (A compiler is one type of translator.)

Rule 7: The control cards $COMPILE, $DATA, $STOP are commands to WATFOR, not to MTS. $COMPILE says "here comes a program for compilation". $DATA denotes the end of the source code and calls for execution of the compiled program; it must be included whether or not you have any input data. $STOP tells WATFOR to turn control back to MTS.

A succession of WATFOR tasks can be run by alternating $COMPILE and $DATA commands as shown in Example 6.

```
$SIGNON MYID '  NAME'
SESAME
$RUN *WATFOR
$COMPILE
   (Source cards for Job #1)
$DATA
   (Data cards for Job #1)
$COMPILE
   (Source cards for Job #2)
$DATA
   (Data cards for Job #2)
$STOP
$SIGNOFF
```

Example 6.

Rule 8: Alternating $COMPILE and $DATA cards is more efficient than executing the WATFOR program separately each time. Note that these tasks are independent. If you have a main program and several subroutines which are to be executed together as a unit, they constitute one task and thus need only one $COMPILE and one $DATA card. In this case, WATFOR distinguishes the end of one routine from the beginning of the next routine by the END card which must be the last card of any WATFOR routine.

Example 7 shows a sample FORTRAN job.

```
$SIGNON MYID '  NAME'
SESAME
$RUN *FTN
     (FORTRAN source cards)
$ENDFILE
$RUN -LOAD
21.67035
$ENDFILE
$SIGNOFF
```

Example 7.

Rule 9: $RUN *FTN performs a similar function to $RUN *WATFOR in Example 5; i.e., it tells MTS to execute the program in the file *FTN, which contains the FORTRAN compiler.

Rule 10: $ENDFILE is used to signify the end of the source code and again to signify the end of the data. It is not a command.

Rule 11: Execution of your compiled program is invoked by the card $RUN -LOAD. This instructs MTS to run the program stored in a file called -LOAD. Fortunately, this is where *FTN has just put your object program since the first RUN command did not specify where the object program was to be stored.

This example points out a major difference between WATFOR and FORTRAN: namely that FORTRAN requires that the translated program be stored somewhere before running it, while WATFOR does not.

AND NOW ANOTHER WORD ABOUT THE MTS COMMAND LANGUAGE

So far, we have encountered several "words" of the MTS command language. There are many more, some of which are discussed in the section on files that follows. In general, the command statements have a fairly free format. A safe rule is:

> Rule 12: Don't leave a space between the dollar sign and the command word. In general, you should leave a space between "words" when there is no explicit separator, such as in $SIGNON MYID. However, if there is a separator such as an equal sign (=), you should not leave any spaces around it. For example, within the "phrases" PW=SESAME and T=30 there should be no blanks.

The experts know abbreviations for the command words, but in general, the first three letters of the command are sufficient. Check to be sure.

Several different tasks can be performed between a single $SIGNON-$SIGNOFF pair. Example 6 showed a sequence of WATFOR tasks; you can have a sequence of WATFOR and FORTRAN tasks; in general, you can have almost any sequence of tasks, whether or not they are related.

THE MYSTERY OF FILES UNVEILED

MTS is a file oriented system that stores files on disk storage. The best way to think of a file is as an area of this disk storage in which you can store information.

Files are composed of lines of information. If information is put in the file via the card reader, then a line is the contents of one card. If the information is put in the file by a FORTRAN program, then generally a line is the information contained in one FORTRAN output record. If the information is put in the file via a typewriter terminal, a line is the information typed before the "return" code is typed. The information contained in a file could be data, an object program, the FORTRAN compiler, a source program, a letter to the President, etc. Every file has a name so that you can communicate with MTS about it.

Some operations that can be done with files are: $CREATE them, and $DESTROY them, $LIST them and $COPY them, $GET them and $RELEASE them, $RUN them, and $EMPTY them. How to carry out these operations is the subject of this section.

Some examples of $RUNning files have already been seen. $RUN *WATFOR commands MTS to execute the program stored in the file called *WATFOR. Since the WATFOR compiler is the program contained in the file named *WATFOR, MTS makes WATFOR start compiling or translating your program. Similarly, $RUN *FTN begins the execution of the FORTRAN compiler in the

file named *FTN. $RUN -LOAD, in Example 7, requests the execution of the program in the file -LOAD. Since the object program has just been stored in -LOAD by *FTN, it is this program that is run.

The most common type of file is one in which every line in the file has a number associated with it. These are called line files and for the purposes of this introduction, when the word "file" is used, we are referring to a line file. The line file is ordered by its line numbers. Thus, regardless of the order in which information was entered into the file (even if you gave line 10 before line 9), the information is effectively stored in the file with the line numbers in ascending order.

There are two ways to specify the line number which is to be associated with a line. One way is to write the numbers yourself at the beginning of each card or typed line. However, files are usually filled initially with a set of sequentially numbered lines, and MTS might as well number them for us via a $NUMBER command shown below.

Creating a File

Example 8 shows a simple batch job that creates a file called FILEFACTS, puts five lines of information into it, and then lists it. Example 9 is the same job but run in conversational mode.

```
$SIGNON MYID '  NAME'
SESAME
$CREATE FILEFACTS
$NUMBER
THIS SAMPLE FILE CONTAINS SOME
INFORMATION ON FILES IN MTS.
1. A FILE IS A SET OF NUMBERED
LINES STORED UNDER SOME UNIQUE
NAME
$UNNUMBER
$LIST FILEFACTS
$SIGNOFF
```

Example 8.

Rule 13: A file can be created by $CREATE followed by the name you wish to attach to it. It is retained for your later use until you $DESTROY it.

Rule 14: The command $NUMBER instructs MTS to number the lines that follow. In batch mode, MTS numbers the lines as they are put in the file. In conversational mode, MTS types the line number for you and then waits for you to type the content of that line. The numbering starts with 1 and goes up by increments of 1 thus giving 1,2,3,... See the $NUMBER command description in the Command Language section of this volume for ways of

specifying a different starting number or different increment. If the $NUMBER command is not given, lines to be put into the file must be numbered explicitly.

Rule 15: Once a file has been $CREATEd, you may enter information into it. MTS must be able to distinguish commands to be executed from text to be stored in a file. All lines of information which (1) start with a line number (either explicitly or via $NUMBER) and (2) do not have a single "$" following the number are put into the currently active file. The currently active file is the file whose name most recently appeared in a $CREATE or $GET statement. ($GET is explained below.) Any line not starting with a line number or containing a single "$" after a line number is assumed to be a command to be executed immediately. If the command is not legal, MTS gives an error comment.

Rule 16: $UNNUMBER turns off the automatic line numbers effected by $NUMBER. MTS still puts any lines with explicit line numbers into the file unless a single "$" follows the line number, or the file is $RELEASEd.

Rule 17: $LIST, followed by a file name, calls for a listing of the contents of the file starting from line 1. If the file has line numbers less than 1, they are not listed. The listing includes the line numbers.

The following example was done on a terminal. The statements typed by the user are in lower case and the output to the user is in upper case. This example illustrates the concept of a prefix character. In an actual session, the area for the password would be blacked out by MTS and "sesame" would not be readable.

```
MTS(L033-0034)
#$signon myid
#ENTER USER PASSWORD
?sesame
#**LAST SIGNON WAS:  11:44.45 02-21-73
# USER "MYID" SIGNED ON AT 12:07.00 ON 02-21-73
#$create filefacts
# FILE "FILEFACTS" HAS BEEN CREATED.
#$number
#     1_  this sample file contains some
#     2_information on files in mts.
#     3_  1. a file is a set of numbered
#     4_lines stored under some unique
#     5_name.
#     6_$unnumber
#$list filefacts
>     1       THIS SAMPLE FILE CONTAINS SOME
>     2     INFORMATION ON FILES IN MTS.
>     3       1. A FILE IS A SET OF NUMBERED
>     4     LINES STORED UNDER SOME UNIQUE
>     5     NAME.
#END OF FILE
#$signoff
#OFF AT 12:08.54    02-21-73
#ELAPSED TIME        1.929 MIN.          $.09
#CPU TIME USED       1.366 SEC.          $.12
#CPU STOR VMI         .116 PAGE-MIN.     $.01
#WAIT STOR VMI        .187 PAGE-HR.
#DRUM READS          53
#APPROX. COST OF THIS RUN IS     $.22
#DISK STORAGE        13.35  PAGE-HR.
#APPROX. REMAINING BALANCE:   $526.76
```

Example 9.

Rule 18: The prefix character is typed by the system and is the first character of every line of output and the first character on a line expecting input. This character identifies who is producing the output line or who is expecting input. Each segment of the system has a distinct character. In this example the "#" appears in MTS command mode and the ">" appears when a listing is being produced. A complete listing of the system prefix characters is given in the "System Command Language" section in this volume.

Revising a File

Suppose that you find after creating a file that you have left out a line in the middle and intended to say more in line 5. You can edit your file on a subsequent run as shown in Example 10.

```
$SIGNON MYID ' NAME'
SESAME
$GET FILEFACTS
2.5, YOU SHOULD LEARN THEM.
5,NAME ASSIGNED BY THE USER.
$SIGNOFF
```

Example 10.

Rule 19: At the start of this run, there is no currently active file. $GET is used to notify MTS of the name of the file to be the active file. $GET FILEFACTS makes FILEFACTS the active file, regardless of whether or not another file had previously been the active file. $RELEASE releases a file from being the currently active file, thus returning to the status of no currently active file.

Rule 20: If $NUMBER isn't used, each line must be numbered explicitly; the number may be separated from the text by a comma. (When a blank is used as a separator, it remains part of the text. A comma, however, used as a separator is stripped off.) If there is no line number, MTS assumes the line is a command to be executed immediately.

Rule 21: A line can be inserted between existing lines by giving it some intermediate number. This number can be fractional, or even negative. (But note that as stated in Rule 17, $LIST still lists starting from Line 1 (unless you take special precautions) even if your file has a line numbered .5.)

A line with a number already in use in the file calls for a replacement of the old line with the new. Thus, in this example, the old line 5 is replaced by the new one.

After running Example 9 and then 10, FILEFACTS has 6 lines: 1,2,2.5,3,4,5.

Copying a File

After a certain amount of editing and insertion of new lines, a file may get rather messy, i.e., full of interpolated lines. For this or other reasons, you may want to copy the information into a new file. Example 11 shows how to do this.

```
$SIGNON MYID '  NAME'
SESAME
$CREATE NEWFILE
$COPY FILEFACTS TO NEWFILE
$SIGNOFF
```

Example 11.

Rule 22: The $COPY command copies the contents of one file into another. In this process the lines are numbered sequentially from 1 as they enter the new file regardless of their line numbers in the old file. The old file is retained unaltered. For ways of copying only part of a file or of altering the numbering in the new file, see the $COPY command description in the "Command Language" section of this volume.

Rule 23: Information that is no longer wanted can be deleted either by a command of the form

$EMPTY FILEFACTS

which removes the contents of the file but retains its identity, or by a command of the form

$DESTROY FILEFACTS

which deletes the file and returns the space on the disk to the pool of available storage.

Most of the time, you will want to use files for programs or for data. Example 12 shows the compilation and execution of a small WATFOR program which is set up in such a way that the source program is left in a file for later use. One method for putting lines that start with "$" into a file is illustrated. An alternate method is shown in Example 17 below.

```
$SIGNON MYID '  NAME'
SESAME
$CREATE PROG
$NUMBER
$$COMPILE
      DIMENSION K(5)
      DO 10 I = 1,5
10    K(I) = (I*5)+7
      STOP
      END
$$DATA
$$STOP
$UNNUMBER
$RUN *WATFOR SCARDS=PROG
$SIGNOFF
```

Example 12.

Rule 24: To prevent a card like $COMPILE from looking like an MTS command for immediate execution, lines beginning with a single "$", whether they are preceded by a line number or not, are not entered into files but are treated as MTS commands. Lines beginning with a line number followed by "$$" are entered into a file with the first "$" removed.

The $RUN card needs some further explanation. WATFOR has to read input--your source program. It must know where to find the input. Is it on a card reader or on a terminal or in a file? And in which of the many card readers, terminals, or files is it? It is undesirable to have WATFOR always expect its input to come from a particular input device. Imagine what would happen if WATFOR insisted that all users input their programs from the third terminal in Room 58 of the Science Library--there would be an intolerable line at that particular terminal and the others would probably be idle. This same problem arises for any program--be it the FORTRAN or WATFOR compiler or your own program which wants to read or write data.

To avoid the necessity of specifying a particular input or output device at the time of writing the program, MTS provides for logical I/O (input-output) units. These are symbolic names and do not refer to actual physical units. The most commonly used logical I/O units are SCARDS, SPRINT, SPUNCH, 0,1,2,...,19. The WATFOR compiler is an example of a program which uses the logical units SCARDS and SPRINT. It reads the source statements from logical unit SCARDS and writes the source listing on logical unit SPRINT. It depends on MTS to tell it at execution time which physical devices are associated with the logical units. It is up to you to give MTS this information.

At the time the $RUN command is given, you must specify, for each logical I/O unit used by the program to be run, which actual physical unit should be used. (In many cases, if you do not specify which device should be attached to the logical I/O unit, MTS makes a plausible assumption. This is discussed below.) Thus, in Example 12, the specification SCARDS=PROG on the $RUN command tells MTS to tell WATFOR that the source program is in a file named PROG. The reason that our earlier examples didn't have to specify SCARDS and SPRINT is that if MTS is not told explicitly, it assumes that SCARDS refers to the device from which the $RUN command came and that SPRINT refers to the device that it has been using to print the command cards.

In the above example, if you wanted your program to read data and had a statement such as READ,X as in Example 5, WATFOR would assume the data could be found in the file PROG since you said SCARDS=PROG. But if you are likely to be varying the data, you would not want to put your data in PROG. You might want to keep your data on cards or in a different file. Example 13 illustrates how to read your data from cards, even when your source program is in a file.

```
$SIGNON MYID '  NAME'
SESAME
$CREATE PROG
$NUMBER
$$COMPILE
      READ(5,1) X
1     FORMAT(2F10.5)
      U=SQRT(X)
      WRITE(6,1) X,U
      STOP
      END
$$DATA
$$STOP
$UNNUMBER
$RUN *WATFOR SCARDS=PROG                -
            5=*SOURCE* 6=*SINK*
21.67035
$SIGNOFF
```

Example 13.

Rule 25: The statements READ(5,1) X and WRITE (6,1) X,U tell WATFOR to use logical unit 5 for input and logical unit 6 for output. Only the numbers 1,...19, are acceptable in these statements. Use of different numbers or of SCARDS, SPRINT, SPUNCH is illegal. This is true in FORTRAN as well as WATFOR.

Rule 26: In the $RUN command, you must tell MTS with what to equate logical units 5 and 6. In example 13, 5=*source* and 6=*sink*, where *SOURCE* and *SINK* are known as pseudo-device names and are explained in the next two paragraphs.

Rule 27: If an MTS command line is too long to fit on a single input line (card or terminal line), a minus sign may be placed in the last column to indicate that the next line is to be considered as a continuation of the first line. The last column of a punched card is column 80; the last column of a terminal line is the last character typed before "return" is signaled.

You want MTS to tell WATFOR where the commands are coming from. If you are in batch mode, logical unit 5 is the card reader. If you are in conversational mode, logical unit 5 is the terminal you are typing on. Similarly, you want 6 to be a line printer or a terminal, depending on what mode you are in. In batch mode, you do not care on which card reader your deck was placed or which line printer you are using. If a particular line printer, for example, is broken or is being used by someone else, you would prefer that MTS gives you a different one instead of waiting for that particular one. If you are using a terminal, you would prefer not to have to worry about finding out some identification number for the terminal at which you are sitting. So you use another gimmick to avoid having to be too specific, the idea of a pseudo-device. Pseudo-devices allow you to take advantage of the fact that, even if you don't know, MTS does know what device your $SIGNON came from and which line printer it can use if you require one.

Wherever you have to name a file or device, you can use a pseudo-device name and let MTS assign a real physical device or file to the name. Pseudo-device names are characterized by the fact that they start with an

asterisk followed by some string of alphabetic characters and end with an asterisk. The most important pseudo-device names which are predefined by MTS are *SOURCE*, *SINK*, *PUNCH*, and *DUMMY*:

SOURCE means that you want the same device from which MTS has been receiving its commands. Thus, in batch mode, *SOURCE* is equated with the particular card reader from which your batch job is being read. In conversational mode, *SOURCE* is assigned to the terminal which you are currently using.

SINK refers to the current output file or device. In batch mode, MTS assigns *SINK* to the line printer assigned to your job. In terminal operations, *SINK* is equated with the terminal you are using.

PUNCH refers to the card punch in both batch and conversational mode.

DUMMY is a name that can be used as an input or output device. If you read from *DUMMY*, you always get just an end-of-file. Used as an output device, it acts like an infinite wastebasket -- you may write as much as you want on it, but it is lost forever.

Example 14 shows how you can put your data into a file called DATA and then run the program that you put in file PROG using the data in the file DATA.

```
$SIGNON MYID '  NAME'
SESAME
$CREATE DATA
$NUMBER
2.173
3.576
4.873
$UNNUMBER
$RUN *WATFOR SCARDS=PROG       -
            5=DATA 6=*SINK*
$SIGNOFF
```

Example 14.

Rule 28: Now logical unit 5 is set to DATA since that is where the program is reading the data from.

Examples 15 and 16 show the corresponding procedure for FORTRAN.

```
$SIGNON MYID '  NAME'
SESAME
$CREATE FPROG
$NUMBER
      READ(5,1)X
1     FORMAT(2F10.5)
      U=SQRT(X)
      WRITE(6,1)X,U
      STOP
      END
$UNNUMBER
$CREATE OBFILE
$RUN *FTN PAR=SOURCE=FPROG     -
           LOAD=OBFILE
$RUN OBFILE 5=*SOURCE* 6=*SINK*
21.67035
$ENDFILE
$SIGNOFF
```

Example 15.

```
$SIGNON MYID '  NAME'
SESAME
$RUN OBFILE 5=*SOURCE* 6=*SINK*
3.1416
$ENDFILE
$SIGNOFF
```

Example 16.

Rule 29: When the source cards are read from a private file, it is not necessary to use a $ENDFILE card to signal the end of the input. MTS automatically supplies an "end of file" indicator when the end of the file is reached.

Rule 30: To tell FORTRAN to put the object program into a private file, one specifies "LOAD=filename" in the "PAR=" field of the $RUN *FTN card. If a file in which to put the object program is not specified, the object program is held temporarily in -LOAD but is then not available for subsequent jobs. By saving the object program as in Example 15, you do not have to recompile the source program in Example 16. Note that this could not be done with *WATFOR since the WATFOR compiler does not provide facilities for retaining the object program.

If you wanted to put the data into a file, you would follow the same procedure as in Example 14.

In Example 12, there is a WATFOR program in a private file PROG. The following example shows a way of getting the same information into PROG without worrying about double dollar signs.

```
$SIGNON MYID '  NAME'
SESAME
$CREATE PROG
$COPY *SOURCE* TO PROG
$COMPILE
      DIMENSION K(5)
      DO 10 I=1,5
10    K(I)=(I*5)+7
      STOP
      END
$DATA
$STOP
$ENDFILE
$RUN *WATFOR SCARDS=PROG
$SIGNOFF
```

Example 17.

Rule 31: The $COPY command tells MTS to copy everything up to the first end-of-file indicator it sees. Since MTS is just looking for the end-of-file mark, lines which ordinarily would be commands requiring execution are copied blindly into the file. When $COPYing from *SOURCE*, the $ENDFILE line serves as the end-of-file indicator.

BUT HERE'S MORE ABOUT FILES

The examples so far have dealt primarily with private files, i.e., files created by the user for his personal use and retained from job to job. There are two other types, temporary (sometimes called scratch) files and public files. A temporary file has a name beginning with a negative sign. You may create and use temporary files just as you do ordinary private files. The only difference is that at $SIGNOFF all temporary files you created are destroyed. As your eligibility for private files is limited, you should use temporary files whenever possible. -LOAD is an example of a temporary file; in this case *FTN automatically creates it for you unless you specify LOAD to be some other file.

Most public files have a name beginning, but not ending, with an asterisk. They may also begin with CCAP:, HELP:, DEMO:, NEW: or OLD:. Anyone may use one, but only the elect may create one. Examples so far have only dealt with public files *WATFOR and *FTN. For a complete listing of all the public files available, see Volume 2 of the MTS manuals, Public File Descriptions. Two more useful public files are *STATUS and *CATALOG. Example 18 shows a run that combines these.

```
$SIGNON MYID '  NAME'
SESAME
$RUN *STATUS
$RUN *CATALOG
$SIGNOFF
```

Example 18.

Rule 32: The program in *STATUS asks MTS to give you a summary of all your computer usage with the ID. You get information such as how much money you have used and how much is left in your account, how much file space you have left, and so on.

Rule 33: The program in *CATALOG lists all the private (permanent) files that you have currently occupying space on disks, that is, all those that you have $CREATEd at some time but never $DESTROYed.

Ordinarily, you are the only one who can read or write on a file which you created. If you want to let other people read your files, you may set a "permit code" that allows it. If, for example, you feel that NEWFILE created in Example 11 ought to be made available to others, you can so specify. Example 19 shows how this is done.

```
$SIGNON MYID '  NAME'
SESAME
$PERMIT NEWFILE ALL
$SIGNOFF
```

Example 19.

Rule 34: After the $PERMIT, you then name the file you want to PERMIT and say what permission is to be extended. ALL means that file may be read by all MTS users. No one but the owner of the file may change it. You can later reverse this effect by issuing the $PERMIT command and specifying NONE after the name of the file instead of ALL.

Now another user can read your file: he can copy it, run it (if it is an object program), or read it as data. Example 20 shows how user ABCD might copy NEWFILE.

```
$SIGNON ABCD ' NAME'
ALIBAB
$CREATE XEROX
$COPY MYID:NEWFILE TO XEROX
$SIGNOFF
```

Example 20.

Rule 35: Reference to files other than your own requires the owner's ID code and a colon (:) immediately preceding the file name. This is necessary since dozens of users may have files called NEWFILE.

To conclude this section, usually input-output devices and files are interchangeable under MTS. You may either think of a file as a substitute for an input-output device, or the reverse. Examples of this interchangeability have been presented in such phases as SCARDS=*SOURCE* which refers to a device and SCARDS=MYFILE which refers to a file.

A LAST WORD CONCERNING TERMINALS VS. BATCH

All of our examples are applicable to batch or conversational modes. The main difference between the two is that in conversational mode, you get immediate feedback from MTS after giving it a command. This can be very helpful if you have made an error. For example, if you misspell the name of a file and MTS cannot find it, it asks you to try another name. If you ask MTS to $DESTROY or $EMPTY a file, it asks you to confirm the request. To do this, you type in either "OK" or "O.K.". If you change your mind, type in anything except OK.

Another difference in conventions between batch and terminal operations occurs in the running of *FTN and *WATFOR. Whereas in batch mode you get a listing of the source program and error comments, on a terminal the source code is normally not listed. Only the erroneous lines and their error messages appear.

Rule 36: If you want your source program listed when running *FTN from a typewriter terminal, add the phrase PAR=SOURCE to your $RUN *FTN line. Note that there are no asterisks in SOURCE. With *WATFOR, to get your source program listed, you must include the parameter SOURCE on your $COMPILE card or line.

When you are entering text into a file, the text begins after the line number and any separator symbol that accompanies it. Thus, if you are entering a FORTRAN or WATFOR program and you want the text to start in column 7, you must give six spaces before you type the line. There are two ways to avoid all the "space-bar hitting". All terminals have tab setting features which may be used. See Volume 4 Terminals and Tapes for the particular terminal being used. In addition, the FORTRAN compiler in *FTN

accepts programs in free-format and the program in file *FORTEDIT may be used to convert your statements from free-format to standard FORTRAN format. (See Volume 2 for descriptions of *FTN and *FORTEDIT.)

If you wish, you can create a batch job from a terminal. If, for example you have used the terminal to debug your program (that is, locate and correct the errors), but you don't want to run the whole program on the terminal, you can request a batch run. Or if you want a listing of your program and would rather use the line printer (which is faster) instead of the Selectric typewriter or CRT (Cathode Ray Tube), you should create a batch job. Example 21 illustrates how this can be done.

```
#$get -temp
#READY.
#$number
#      1_$$signon myid '  oscar'
#      2_sesame
#      3_$$run *ftn par=source=p
#      4_$$run -load
#      5_$$signoff
#      6_$unnumber
#$list -temp
>      1      $SIGNON MYID ' OSCAR'
>      2      SESAME
>      3      $RUN *FTN PAR=SOURCE=P
>      4      $RUN -LOAD
>      5      $SIGNOFF
#END OF FILE
#$copy -temp *batch*
>*BATCH* ASSIGNED RECEIPT NUMBER
603772
>*BATCH* 603772 RELEASED
#$signoff
#OFF AT 12:34.50    02-21-73
#ELAPSED TIME        2.632 MIN.
#CPU TIME USED       1.516 SEC.
#CPU STOR VMI         .15  PAGE-MIN
#WAIT STOR VMI        .286 PAGE-HR.
#DRUM READS          51
#APPROX. COST OF THIS RUN IS  $.37
#DISK STORAGE        15.9   PAGE-HR.
#APPROX. REMAINING BALANCE:  $100.
```

Example 21.

Rule 37: To create a temporary file, you merely have to mention the name. Thus, in this example -TEMP is created implicitly in the $GET statement.

Rule 38: To run a batch job from a terminal, copy a file, in this case -TEMP, to *BATCH*. -TEMP must contain a complete batch job including $SIGNON and $SIGNOFF. A six-digit receipt number is typed which must be presented at the Production Control Desk to collect the job (unless it is delivered). If the same sign-on is used for both the batch job and the present terminal session, the batch job will be run as soon as possible after the signoff at the terminal.

Rule 39: Note that when you $SIGNOFF, file -TEMP is destroyed since it is only a scratch file. This, however, is no problem since *BATCH* already has all the information it needs from -TEMP. If you expect to use this sequence of commands again and would like to have them saved, you should use a permanent file instead of a scratch file.

All that remains to be said is:

```
$GOOD LUCK
$RUN TO CONSULTANTS IN ROOM 333, ASB 2.
```

April 1974

A BRIEF OVERVIEW OF MTS

ACCESS TO THE SYSTEM

All access to MTS is controlled by the system itself from information supplied to it by the CDPC administrative staff. See Facilities and Services of WSU Computing and Data Processing Center for Academic Users (February 1973) for information on how to apply to use MTS.

As its name implies, the Michigan Terminal System was designed for access from remote terminals, although the system also has extensive facilities for processing batch jobs.

Batch and terminal access differ from each other as little as possible. The command language is identical, and the same translators and utility programs are available in both modes. Batch, however, is usually inappropriate for jobs requiring interaction between the user and the program.

Although some users need only batch access and others only terminal access, the majority of users use both: terminals for program development and debugging, and batch for bulk-data input and output, and production running (for non-interactive programs).

Batch may be either local or remote. Local batch is submitted and retrieved at the CDPC. The mechanics of this are described in the manuals titled Facilities and Services of WSU Computing and Data Processing Center for Academic Users (February 1973) and Introduction to MTS @ WSU (January 1974); internal processing and options of interest to users submitting batch jobs are described in the section "Batch Usage of MTS". At the remote batch (often called Remote Job Entry, or RJE) station, batch card decks are read and printed output is produced from information transmitted via telephone lines to and from the Computing Center. The operational details of this remote station are described in "Batch Usage of MTS".

Terminal access depends on the type of terminal being used. The following terminal types are currently supported by MTS. (This is not an exhaustive list; see "Conversational Usage of MTS" later in this volume.)

Selectric typewriter terminals
IBM 1050 (and 1056 Card Reader)
Model 33 Teletype
Model 35 Teletype
Westinghouse 1600 Cathode Ray Terminal

Any terminals compatible with the Selectric typewriter terminals or the Teletype (such as IBM 2741 and DURA) are also supported. All terminals are connected by means of the dial-up facilities of the telephone company. The

system is accessed through the MEMOREX Transmission Control Unit. Details may be found in the appropriate sections of the "MEMOREX 1270 Device Commands" in Volume 4.

DATA INPUT AND OUTPUT

Batch users may enter input data via punched cards and receive output via printed paper or punched cards. Magnetic tapes may be used by both batch and terminal users. Details on programming and usage of magnetic tapes are found in "The Magnetic Tape User's Guide" in Volume 4.

GENERAL CONCEPTS

User programs, system translators, and utility programs are all treated identically; they are just programs. The command "$RUN xx" is used to request execution of the object program contained in the file "xx". Thus, the command $RUN *FTN requests MTS to execute the FORTRAN compiler, and $RUN OBJ requests MTS to execute the user's object module in the file OBJ.

Logical I/O Units

Programs obtain data and produce output by means of logical I/O units. For example, a translator typically reads the source program from logical I/O unit SCARDS, produces listing output on logical unit SPRINT, and writes the translated object program on SPUNCH. When the user invokes a program using the $RUN command, he specifies a file-or-device name (FDname) to be attached to each of the logical units the program will use. For example,

```
$RUN *WATFOR SCARDS=FYLA
```

attaches file FYLA to SCARDS, so that WATFOR reads from FYLA which hopefully contains a source program. If a logical unit is not explicitly assigned, a default is usually taken. File-or-device names can be:

file names	FYLA	
device names	>RDR1	
pseudo-device names	*SOURCE*	Specifies the input card deck for batch, the keyboard for terminals.
any of the above with line number ranges	FYLA(10,15)	Specifies all lines in FYLA with line numbers in the range 10 through 15.

any of the above with modifiers	FYLA@¬TRIM	Specifies no trimming of trailing blanks.
or any combination of the above (called explicit concatenation)	FYLA(10,15)+FYLB+C@¬TRIM(1,1000)	

Details of the names that can be specified and their meanings are found in the section "Files and Devices".

Prefix Characters

When a user is at a terminal, a special character is printed as the first character of every line of output and as a first character on the line before waiting for input. This character is called the prefix character and identifies the system component or program which is writing the output line or waiting to read an input line. Each component of the system has a distinct character. For example, "#" in front of an input line means that the user is in MTS command mode. When a program is running, the prefix character is normally a blank. (The user may change this by calling the subroutine SETPFX. See the subroutine description in Volume 3.) A complete list of system prefix characters is given later in the section "System Command Language".

Virtual Memory

One of the primary constraints facing programmers has been the amount of addressable memory or main storage available. Addressable memory (sometimes called core storage, for historical reasons), as opposed to auxiliary storage, refers to storage that can be addressed directly. In the past, when a program required more than the maximum main storage available, the "ping-pong" or "overlay" method was often used. Using this technique, the programmer divided his program into self-contained sections, each small enough to fit into main storage. Each section was loaded and executed separately, and it was up to the programmer to control the order, loading, and linking of the sections. This was no simple task.

The concept of virtual memory, implemented in the IBM 360/67, allows the user more addressable memory than is physically available. In effect, the system is doing the "ping-ponging" for the programmer and he does not have to worry about any of the details. The system breaks the program into sections called pages, "paging" in sections when they are needed and "paging" them out when they are no longer needed.

Virtual memory is many times larger than actual physical memory and is divided into segments. A segment is 256 pages and a page is 4096 bytes

(characters). There are 1,048,576 bytes in one segment. Each user may use several (currently up to four) segments, besides the three segments which are shared by all users. These three segments contain the re-entrant routines that are commonly used, such as the resident system, MTS, and the I/O routines. Great economy is achieved by keeping only one copy of each of these routines in memory instead of one for each user.

The primary advantage of the virtual memory concept is apparent--each user has access to a very large address space. The system insures that when a page is needed, it is brought into main storage if it is not already present. In order to utilize the system efficiently, it is desirable to minimize the number of different pages needed in rapid succession. The following programming suggestions can reduce the amount of paging required.

In FORTRAN, when indexing a large multi-dimensional array which requires more than one page, it is advisable to vary the left-most subscript most rapidly. This causes the array elements to be accessed in the order that FORTRAN stores them.

When searching a table, a linear search through one page is more efficient than a random search through several pages. When a table is larger than a page and has many pointers, it is better to use an index table to get directly to the proper page rather than to follow all the pointers in succession.

MTS COMMAND LANGUAGE

In order to use the computer, one must first get the attention of MTS and identify himself through the $SIGNON command. Once he has done this and MTS has agreed that he is a legitimate user, the IBM 360/67 is at his disposal.

In this section, the commands are listed in their most common form followed by a brief explanation of their function. Not every possible option and parameter is mentioned. A detailed explanation of the commands may be found in the section "System Command Language".

As stated earlier, command lines start with a "$" followed immediately by the command name or the command abbreviation. The "$" is required for batch runs but not necessarily for conversational runs. In a conversational interaction with MTS, the "$" for a command is not needed if (1) there is no line number at the front of the input line, and (2) automatic line numbering is off or there is no currently active file in which to place data lines. Under these conditions, the command line may start with the command name or the command abbreviation. If these conditions do not hold, then the leading "$" is required on all MTS commands. Note that these restrictions essentially say that MTS attempts to interpret an input line as a command line, even without a leading "$", if it is not prepared to accept a data line from the user and if the user is in conversational mode.

A. Global Control

$SIGNON yyyy	tells MTS that the user with ID "yyyy" wants to use the machine. The password associated with the ID must be entered either with the $SIGNON command or on the following line.
$SIGNOFF	signs the user off the machine and gives him a summary of the cost of the run.
$SET	is used to set various switches and variables in the system. These control such functions as setting the user's password, automatic errordumping, the line number separator character, upper- and lower-case conversion, implicit concatenation, plus a few more esoteric switches.
$SINK xx	makes the pseudo-device *SINK* refer to the file or device "xx" instead of the printer (in batch mode) or the terminal on which the user is typing (in conversational mode).
$SOURCE xx	makes the pseudo-device *SOURCE* refer to the file or device "xx" instead of the card reader (in batch mode) or the terminal on which the user is typing (in conversational mode).

B. Program Control

$RUN xx	tells MTS to load and execute the program in the file "xx". These files can be public, such as *FTN or *STATUS, or private, such as MYPROGR or -LOAD.
$LOAD xx	tells MTS to load but not execute the program in the file "xx". The program can then be displayed and/or altered. Execution can be started by the $START command.
$START	starts the execution of a program which has already been loaded by a $LOAD command.
$RESTART	restarts the program at the point of the last interrupt or at a specified location. $START and $RESTART are synonymous and may be used interchangeably.
$UNLOAD	releases storage and devices from the previous $LOAD or $RUN. It is useful when the execution of a program did not terminate normally.

C. For Debugging

$DEBUG xx	tells MTS to put the Symbolic Debugging System (SDS) in control. SDS has its own set of commands which allows the user to interactively debug his program. A listing of the important debug commands is given below.
$SDS	returns control to SDS from MTS.
$ALTER xx yy	allows the user to change the contents of a general register, a floating point register, or a specified location in virtual memory. "yy" becomes the contents of "xx".
$MODIFY	is a synonym for $ALTER.
$DISPLAY xx	displays the contents of "xx", where "xx" may be a general register, floating-point register, or a specified region of virtual memory.
$HEXADD xx yy	prints out the hexadecimal sum of the hexadecimal numbers "xx" and "yy".
$HEXSUB xx yy	prints out the hexadecimal difference between the hexadecimal numbers "xx" and "yy".
$DUMP	prints out the contents of the general registers, floating-point registers, and all the virtual memory locations associated with the current job.
$ERRORDUMP	provides a dump of all the registers and storage if the execution of a program terminates abnormally. This command is effective in batch mode only. (This is the same as $SET ERRORDUMP=ON.)

D. File Handling

$CREATE xx	creates a file named "xx". If it is not a temporary file (that is, if its name does not begin with a minus sign), it is retained until the creator destroys it. "xx" becomes the currently active file.
$DESTROY xx	destroys the file named "xx" and returns the space on disk to the pool of available storage.

$EMPTY xx	empties the contents of the file "xx" but retains its identity. Thus the file can be reused later.
$COPY xx TO yy	copies the contents of the file "xx" into the file "yy". Unless specified otherwise, the lines are numbered sequentially from 1 as they are entered in the new file regardless of their numbers in the old file. The old file is retained unaltered.
$LIST xx	lists (with line numbers) the contents of file "xx" on the line printer (batch mode) or the terminal (conversational mode).
$GET xx	makes "xx" the currently active file. If "xx" is a private file, it must have been created previously. If "xx" is a temporary file and does not already exist, it will be implicitly created by this statement.
$RELEASE xx	releases the currently active file, dismounts a magnetic tape "xx", or releases a HASP pseudo-device "xx".
$NUMBER	tells MTS to automatically number the lines that follow as they are put in the currently active file. Unless otherwise specified, numbering starts with 1 and increments by 1.
$UNNUMBER	tells MTS to stop the automatic numbering which one presumably requested earlier by a $NUMBER command.
$EDIT xx	tells MTS to put the EDITOR in charge. Like SDS, the EDITOR has its own set of commands which allow the editing of the file "xx". A listing of the EDITOR commands is given below.
$PERMIT xx	changes the access code of file "xx" to either allow or not allow other users access to file "xx".

E. Miscellaneous

$COMMENT	allows insertion of comments on output to the terminal or printer.
$CALC exp	evaluates the arithmetic expression "exp" and prints the result.
$CONTROL xx yy	sends the control command "yy" to the pseudo-device "xx".

$INQUIRE xx	displays the current status of the batch job, an execution queue, a print queue, a punch queue, plotting jobs, OS batch jobs, UMMPS jobs, or other system activity.
$MOUNT xx	mounts the magnetic tape or MERIT network connection "xx".
$NET xx	enters into the MERIT communications network system for the connection "xx".

THE LINE FILE EDITOR COMMAND LANGUAGE

After the user has entered the command $EDIT xx, MTS turns control over to the MTS line file editor which is ready to accept commands concerning the editing of the file named "xx".

A user editing a file may edit either by line number or by context.

The editor maintains a current line pointer which is initially set to the first line of the file. Commands that require line number specification may refer to the line pointer or may explicitly mention the appropriate lines.

This section gives a brief description of the command language understood by the file editor. For the complete description of the line file editor, see the "Edit Mode" description in the "System Command Language" section.

A. Commands Which Move the Line Pointers

LINE n	causes line "n" to become the current line.
SCAN	scans a specified area of the file for a particular string and sets the line pointer to the line containing the string.
+n or n	moves the line pointer forward "n" lines.
-n	moves the line pointer backward "n" lines.

B. Commands Which Change Characters within a Line

ALTER	allows the user to replace one string of characters with another in a specified line.

BLANK	allows the user to blank out parts of a line selectively.
OVERLAY	allows the user to replace selected characters in the old line.
SHIFT	shifts characters right or left within a line and inserts blanks into vacated positions.

C. Commands Which Change Entire Lines

COPY	copies specified lines (without deleting them) to another region of the same file.
DELETE	deletes specified lines from the file.
INSERT	allows the user to insert new lines into the file.
REPLACE	causes the lines in the specified range to be replaced by the given string.

D. Commands to Get Out of EDIT Mode

MTS	returns control to MTS. To return to the editor, use the EDIT command.
STOP	causes the file editor to return control to MTS.

E. Commands Which Affect Editor Control

EDIT xx	tells the editor that a new file named "xx" is now to be edited.
SET	is used to change the values of a number of switches which govern the editing process.
REGION /name	allows the user to name subsections of a file with a region name "/name", for subsequent reference.
XEC $name	allows the user to store a sequence of editor commands as a program named "$name".

F. Miscellaneous

CHECKPOINT	tells the editor to keep a copy of the file as it is at the time of this call so that it may be restored later, if necessary.
RESTORE	is used in conjunction with CHECKPOINT and tells the editor to restore the specified lines (or the whole file) to their condition at the time of the last CHECKPOINT command.
EXPLAIN	asks the editor to explain its entire operation, a particular command, or an error comment.
PRINT	requests the editor to print out certain parts of the file.
RENUMBER	commands the editor to renumber the file, where the user can specify the starting number and increment.
COLUMN	specifies the column range that can be referenced by the commands ALTER, MATCH, SCAN, and SHIFT. Initially the range is from 1 to 255.

DEBUG COMMAND LANGUAGE

The Symbolic Debugging System (SDS) is a conversational program checkout facility which aids in the process of debugging programs from a remote terminal. The MTS command $DEBUG xx has the same format as the $RUN command. It tells MTS to load the program in file "xx" and then give control to SDS. Using the SDS command language, the programmer initiates execution of his program and monitors its flow by specifying breakpoints where instructions and data may be modified and displayed.

The programmer may refer to locations in his program symbolically, by relative address, or by absolute (virtual or loaded) address. Symbolic referencing can be used only with those language processors which generate a symbol table with the object programs they produce. (Currently, this includes the G-level 360 assembler and FORTRAN-G.) If the user has requested that SDS display the contents of a particular location--using any of the above three ways to name the location--SDS consults the symbol table, if there is one, to get the proper type and length of the location. If there is no symbol table, the contents are printed in hexadecimal in units of 4 bytes.

This section gives a brief description of the more common commands available. For a complete description of all the debug commands, see the "Debug Mode" description in the "System Command Language" section.

A. Setting and Clearing Breakpoints

AT xx	inserts a breakpoint in location "xx" of the program being debugged. When execution reaches location "xx", the SDS commands following the AT command are executed.
END	marks the end of a sequence of commands started by an AT command.
BREAK xx	inserts a breakpoint at location "xx". When the program being debugged attempts to execute the instruction at location "xx", control is given to SDS which then requests further commands from the user.
CLEAN	deletes all breakpoints set by the AT and/or BREAK commands.
RESTORE xx	deletes the breakpoint "xx" or the most recently created breakpoint if "xx" is not specified.

B. Control Commands

RUN	transfers control to the entry point of the program being debugged, thus starting execution.
CONTINUE	causes the program being debugged to continue execution after a breakpoint or an interrupt has occurred.
GOTO xx	transfers control to location "xx" of the program being debugged.
STEP n	allows the user to step through the next n instructions.
MTS	returns control to MTS. To return to SDS, use the MTS $SDS command.
SDS	transfers control to SDS which then requests more commands. This command is used with the AT command.
STOP	stops SDS processing, unloads the user's program, and returns control to MTS.

C. The Actual Debugging Commands

MODIFY xx yy	changes the contents of location "xx" to contain "yy".
ATTRIBUTE yy	prints out the attributes, such as length and address, of the symbol "yy".
DISPLAY xx	prints the contents of location "xx".
HEXDISPLAY xx	prints the contents of location "xx" in hexadecimal.
SCAN xx yy	scans through the region "xx" for the value "yy". If the search is successful, the location where it was found is printed.
SYMBOL xx	prints the symbolic name of location "xx".
DUMP	prints a symbolic dump of the user's program.

D. Setting SDS Parameters

SET	allows the user to set various control parameters in SDS. These include the default length, scale and type attributes, the processing of program and attention interrupts and the processing of dynamically loaded sections.
RESET	allows the user to reset the SDS control parameters.
CSECT xx	makes control section "xx" the current section.
USING	is used when the program being debugged employs DSECTs. The command provides addressability for DSECTs.

E. Miscellaneous

MAP	produces a map of all the sections in the program which were processed by SDS.

April 1974

LANGUAGE TRANSLATORS

MTS provides a wide variety of translators for assorted programming languages. This section gives a brief description of the languages available and mentions which public files contain the appropriate translator. Volume 2 describes all public files containing the language translators. These descriptions should be checked before actual usage is attempted.

Procedure-Oriented Languages

ALGOL is a procedure-oriented language useful for numerical calculations. The processor is located in the public file *ALGOL.

ALGOLW is a revised version of ALGOL. This is a revision of Stanford University's ALGOL W which was done by the University of Newcastle. This processor is in the file *ALGOLW.

COBOL (Common Business Oriented Language) is a programming language suitable for commercial data processing work. The file *COBOL contains a version of the IBM ANS COBOL compiler.

FORTRAN IV (Formula Translator) is a language used for solving problems which can be stated in mathematical terms. The IBM G-level FORTRAN compiler is available in the public file *FTN. The IBM H-level FORTRAN compiler--which is slower and more expensive than the G-level compiler, but produces shorter object programs and better error comments--can be found in the file *FORTRANH.

PL/I (Programming Language I) is a programming language which is suitable for both scientific and business computations. The compiler is available in the file *PL1.

SPL (Student Programming Language) is primarily a subset of PL/I and is useful for students in introductory courses. The compiler is available in the file *SPL.

WATFOR (Waterloo FORTRAN) is a load-and-go (compile-and-execute) FORTRAN compiler useful for running short debugging jobs in the FORTRAN language. The compiler has excellent compile-time and run-time diagnostics, but programs run under WATFOR execute more slowly than those run from object decks produced by the FORTRAN compiler. If the user intends to execute the program often, he is advised to debug the program using WATFOR and, after it is debugged, to obtain an object deck from *FTN. The WATFOR compiler is available in the files *WATFOR and *SWAT. (The version in *SWAT is for shorter programs.)

XPL is a language useful for writing compilers and can be found in the file *XPL.

Assemblers

IBM 360 Assembler Language is a low-level (one step removed from machine language) symbolic language. The IBM G-level Assembler is available in the file *ASMG.

IBM 1130 and 1800 Assembler Language programs may be assembled by the assembler located in the file *1130ASM.

PDP-x Assembler Language facilities are available for those programmers desiring object decks for use with Digital Equipment Corporation PDP-1, PDP-5, PDP-7, PDP-8, or PDP-9 machines. The file *1ASR has the assembler for PDP-1 machines; file *8ASR has the assembler for PDP-5 and PDP-8 machines; the file *9ASR has the assembler for PDP-7 and PDP-9 machines.

PL360 is an ALGOL-like translator which gives the source language programmer much of the power and flexibility of writing assembly code. The translator is available in the file *PL360.

STASS 360 (Student Assembler 360) is an assembler especially tailored for student use. It provides fast batch assembly and execution of small jobs with extensive error-checking at both assembly and execution time. This assembler can be found in the file *STASS360.

Interactive Languages

APL (A Programming Language) is a conversational programming language which is best suited for problems requiring extensive manipulation of a small amount of data. It can be run from Selectric typewriter terminals and requires a special type ball for the terminal. APL is available in the file *APL.

BASIC is an easy-to-use interactive language which operates in its own subsystem. Within BASIC one can do numerical calculations as well as simple MTS-like functions such as file manipulations. The BASIC system is located in the file *BASIC. A full description of BASIC is given in Volume 10.

PIL (Pittsburgh Interpretive Language) is an interactive, procedure-oriented language particularly designed for terminal use. The interpreter is available in the file *PIL. A newer version of PIL is documented in MTS Volume 12 and is located in the file CCAP:PIL2.

REDUCE is designed for general algebraic computations of interest to physicists and engineers. Its capabilities include: expansion and ordering of rational functions of polynomials, symbolic differentiation, reduction of quotients of polynomials by cancellation of common factors, calculation of symbolic determinants, and calculations of interest to high-energy physicists, including spin 1/2 and spin 1 algebra. The program is written in LISP and may be accessed by $SOURCE *REDUCE2.

List- and String-Processing Languages

LISP is a list-processing language available in the file *LISP.

SLIP (Symmetric List Processor) provides list-processing capabilities. MTS has provided these capabilities in the form of a library of FORTRAN-callable subroutines. The library is in *SLIP and must be concatenated to the object modules which contain the subroutine calls.

SNAP is an English-like language designed for those persons who perform text-processing tasks but desire minimal computer knowledge. The processor for SNAP is available in the file *SNAP.

SNOBOL4 (String Oriented Symbolic Languages) is a symbol- (string-) manipulation language which is useful in language translation, program compilation, and combinatorial problems. A compiler for SNOBOL4 is available in the file *SNOBOL4. The blocks version of SNOBOL4 has additional features, making it especially good for text layouts. This version may be found in the file *SNOBOL4B. A faster version of SNOBOL4 may be found in *SPITBOL.

UMIST (University of Michigan Interpretive String Translator) 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. The translator can be found in the file *UMIST.

*1 is an L-6 type language for processing list structures at the assembly language level. It is implemented in the form of macros callable from the 360 Assembler Language found in *ASMG. The macros are in the file *1. For details on the usage of this library, see the *1 public file description in Volume 2.

Simulation Languages

CSMP (Continuous System Modeling Program) is a simulation language specially designed for those simulation models which are expressed

as differential equations or as equivalent analog block diagrams. The translator is in the file CCAP:CSMPTRAN, the executor is in the file CCAP:CSMPEXEC, and the library to be concatenated at run time is in the file CCAP:CSMPLIB.

GPSS (General Purpose Simulation System) is a program for conducting discrete event evaluations of systems, methods, processes, and designs. The main feature of GPSS is a simple flow chart language for describing the problem or system to be simulated. Once this description is read into GPSS, the program automatically performs the simulation including the gathering of statistics. The results may be printed out automatically or under control of the user, who may wish to present them in a graphical format. The program is available in the file CCAP:GPSS.

SIMSCRIPT-II is a general programming language that has features of particular interest for discrete event simulation problems; however, the language is rich enough to be used also in non-simulation problems. The compiler is available in the file CCAP:SIM2. The library to be concatenated at run time is in the file CCAP:SIMLIB.

Statistical Program Packages

SPSS (Statistical Package for the Social Sciences) is an integrated set of programs for the description and statistical analysis of social science data. SPSS contains routines for the preparation and editing of data, for the calculation and display of frequency and joint frequency distributions, as well as a large number of parametric and non-parametric statistical procedures often used by social scientists. The program is available in the file CCAP:SPSS. Manuals describing its use are available at the WSU Bookstore.

CONSTAT (CONsole STATistics) is a console-oriented (user prompting) statistical computing program. It sets up a data structure which is common to many statistical problems and provides a set of of subroutines to perform statistical analyses on the data. The program is available in the file CCAP:CONSTAT. Documentation is available at the WSU Bookstore.

OSIRIS (Organized Set of Integrated Routines for Investigation with Statistics) is a set of programs for the analysis of social science (especially survey) data. Data management includes generation of specially formatted files, and corrections and transformations of these files. The statistical part includes multivariate and non-parametric analysis programs. Manuals describing its use are available in the bookstore.

Information Retrieval

BIRS (Basic Information Retrieval System) is a set of programs for the creation of data bases and for the retrieval and reporting of information extracted according to the user's specification. These reports may take the form of printed indexes and catalogs. A search language permits file queries using standard logical operators. The program is available in the file CCAP:BIRS. Documentation is available at the WSU Bookstore.

SERVICE FILES

MTS provides a number of public files which can best be described as service or information files. The most useful of these are described in this section.

Information about a Particular ID Number

$RUN *STATUS prints out all the statistics of the user SIGNON ID given on the $SIGNON command. The statistics include such items as the amount of money spent and remaining, the disk storage currently used and remaining, the number of terminal and batch jobs run, and so on.

$RUN *CATALOG gives a listing of all the user's files currently defined.

$RUN *FILESNIFF allows the user to get detailed information about the status of a particular private file.

Information about Activities of the Computing and Data Processing Center

$COPY HELP:DELIVERY lists a file of all the CDPC delivery codes.

$COPY HELP:DIRECTORY lists a file of references to other information sources organized by subject.

$COPY HELP:HELP lists a file of names of people who are willing and able to provide help in the use of some (or all) aspects of the computing facilities.

$COPY HELP:RATES lists a file of charges for various CDPC services.

$COPY HELP:HOURS lists a file containing the schedule of various CDPC services.

$COPY HELP:NEWS lists a file of current news from the CDPC. Entries are listed in reverse chronological order.

Documentation Aids

Two programs exist which are useful in formatting and editing documents. The program in the file *FMT is good for short documents. TEXT360, good for longer documents, has several components, each residing in the separate files *PRESCAN, *FMAINT, *BLDLIN, *PLAOUT, *POSTPR and *PRINT.

Information on these programs is found in Volume 5.

April 1974

INTRODUCTION TO THE EDITOR

The context editor is a useful tool for either changing the contents of a file or searching a file for certain items. Entire lines can be inserted, deleted, or replaced, or changes can be made to only part of a line.

In the following discussion, assume that the file PROG is a line file which contains a rather botched program. The examples are given exactly as they would appear on a terminal. Input from the user is in lower case; output from the editor is in upper case.

To enter edit mode from MTS command mode, use the EDIT command specifying the file to be edited as the parameter:

```
#edit prog
:
```

Note the change in prefix characters above. Before entering the EDIT command, the prefix character was "#" indicating that we were in MTS command mode. Now the prefix character is ":" indicating that we are now in edit mode and communicating with the editor.

Editor commands begin with an edit command name. For example, to print the first line of the file:

```
:print
:      1            DIMENSION DATA(50)
:
```

The "1" at the front of the line is the line number of that line in the file.

Modifiers may be appended to the command name to change the behavior of the command. For example, to print the first line without the line number, append the "@NL" modifier (all modifiers start with an "@"):

```
:print@nl
:      DIMENSION DATA(50)
:
```

Command names can be abbreviated. In all cases, the first three letters are enough. Sometimes only one or two letters is enough. For example:

```
:p
:      1            DIMENSION DATA(50)
:
```

Unless one is certain that a given abbreviation represents the command wanted, rather than another command, it is wise to use the full name. All further examples use the full command name.

The first parameter on any edit command that refers to the file specifies what line or lines of the file the command is to work on. If this parameter is omitted, only the current line is used (except for two commands which search the file: SCAN and MATCH). Thus, in the above PRINT example, the current line (which was initially set to be the first line of the file) was used. To specify a different line, include the line number, for example:

```
:print 3
:      3            REAL MEAN
```

One number, as above, means a single line. To specify a range of lines, give the beginning and ending line numbers of the range:

```
:print 3 5
:      3            REAL MEAN
:      4            WRITE (6,100)
:      5       100  FORMAT(' ENTER NUMBER OF DATA POINTS')
:
```

Three special symbols can be used in place of a line number:

```
*L  represents the number of the last line in the file
*F  represents the number of the first line in the file
*   represents the number of the current line
```

For example to print the last line specify:

```
:print *1
:      28           END
:
```

To perform an operation on all lines of the file, the symbol "/FILE" may be used to represent the line number range that covers the entire file.

In order to develop examples of editor usage, a copy of the original file is needed:

```
:print /file
:       1           DIMENSION DATA(50)
:       2           COMMON DATA,N
:       3           REAL MEAN
:       4           WRITE (6,100)
:       5     100   FORMAT(' ENTEF NUMBER OF DATA POINTS')
:       6           READ (5,101) N
:       7     101   FORMAT(13
:       8           WRITE (6,102)
:       9     102   FOR.MAT(' ENTER DATA POINTS')
:      10           READ (5,103)(DATA(I),I=1,N)
:      11     103   FORMAT(6F3.2)
:      12           CALL CALC(MEAN,STD)
:      13           WRITE (6,104) NEA
:      14           WRITE (6,104) MEAN,S
:      15     104   FORMAT(' MEAN=',F8.4,' STD='F8.4)
:      16           END
:      17           SUBROUTINE CALC(MEAN,STD)
:      18           DIMENSION DATA(50)
:      19           COMMON DATA,N
:      20           REAL MEAN,MEAN2
:      21           DO 10 I=1,N
:      22           X = X+DATA(I)
:      23     10    Y = Y+DATA(I)*2
:      24           MEAN = X/N
:      25           MEAN2 = Y/N-MEAN**2
:      26           STD = SQRT(MEAN2)
:      27           RETURN
:      28           END
```

The remainder of this section is an explanation of how to fix up the file (or at least improve its condition).

There are two basic methods by which one can use the editor. One method is by line number. In this method one needs a listing of the file with the line numbers, and then the editing is done by explicitly specifying the line number. Since we have the listing of the file, this is the method we used in this introduction. The other method is by context. In that method one uses the SCAN and MATCH commands which search the file for the lines to be changed. That method is often used, for example, when one discovers that he has used both the name MEAN and the name AVER for the same thing, and he wants to go through the file change all occurrences of AVER to MEAN.

DELETE, INSERT and REPLACE are commands that work on an entire line as opposed to altering part of a line.

The DELETE command deletes either a single line or a range of lines. It is used in the same manner as the PRINT command. In the file PROG, note that line 13 is extraneous and should be deleted.

```
:delete 13
:
```

However, it is very easy to think "13" while actually typing "23", and consequently, the wrong line is deleted. A safer procedure would be to set the value of the current line to the line number of the line to be deleted. The editor then prints the current line. After checking that it is correct, the DELETE command is given without a parameter to delete the current line.

```
:line 13
:      13          WRITE (6,104) NEA
:delete
:
```

The INSERT command is used to insert a line or group of lines in between existing lines of the file. INSERT never replaces or moves existing lines. If there is no "line number room" to insert another line, because the two existing line numbers differ by only .001, the comment "NO ROOM FOR FURTHER INSERTION" is given and it is up to the user to renumber the file or move lines around to open up space.

There are two forms for the INSERT command. To insert only one line, specify it as a string on the command. To specify a string, give a delimiter, followed by the characters of the string, and terminate with the same delimiter. The delimiter cannot be a letter or a digit and cannot appear in the characters of the string itself. For purposes of this introduction ";", ":", and "#" are used because they are legal, because they do not usually appear in FORTRAN programs, and because at least one of them is a convenient lower-case character on most terminals. (On Teletypes, ":" and ";" are lower case; on Selectric typewriter terminals, "#" is lower case.) Other characters are legal; see the "Edit Mode" description for details.

The line number specified in the INSERT command specifies the line after which the new line or lines are to be inserted. If no number is specified, the new line or lines are inserted between the current line (as specified by the last LINE command) and the following line. Thus, to insert a "GO TO 1" statement before the END statement of the main program, specify:

```
:insert 15 ;          go to 1;
:      15.25         GO TO 1
```

Or use the safer procedure and set the current line to 15, check to see that it is the the right line, and then insert after line 15:

```
:line 15
:      15        104 FORMAT(' MEAN=',F8.4,' STD=',F8.4)
:insert #        go to 1#
:      15.25          GO TO 1
:
```

Note that in both cases the insertion is verified by printing the new line contents. Although this verification can be suppressed, it is strongly suggested that it be left on and the line be inspected to verify that the correct insertion was made.

To insert more than one line use the second form of the INSERT command. By leaving off the string parameter when specifying an INSERT command, the editor enters "fast-insert" mode in which the editor reads lines from the terminal to be inserted into the file. To leave "fast-insert" mode, enter a null line or an end-of-file condition. Thus, to initialize the variables X and Y in the subroutine CALC in our file, specify:

```
:insert 20
?       x = 0.0
?       y = 0.0
?
:
```

Or else specify:

```
:line 20
:       20              REAL MEAN,MEAN2
:insert
?       x = 0.0
?       y = 0.0
?
:
```

Note that the editor switches the prefix character to a "?" to indicate that all lines entered are going into the file. It is a common mistake to enter "fast-insert" mode, enter the lines wanted, forget about leaving "fast-insert" mode, and then to enter an edit command which is promptly inserted into the file. Watch the prefix character!

The REPLACE command replaces one line with a new line which is supplied in one of two forms. The first is exactly like the first form of the INSERT command, where the new line is specified as a string. Thus, line 7 is fixed by replacing the whole line. (This is not the recommended way to fix it -- later a better method is demonstrated.)

```
:replace 7 # 101  format (i3)#
:       7       101  FORMAT(I3)
:
```

Or else:

```
:line 7
:       7       101  FORMAT(I3
:replace ; 101  format(i3);
:       7       101  FORMAT(I3)
```

For the second form of the REPLACE command, if the new line is not specified as a string on the command, the editor prints the old contents of the line and then prompts for the new contents:

```
:replace 7
: 101  FORMAT(I3
? 101  format(i3)
```

```
:       7       101  FORMAT(I3)
:
```

Now consider the commands that change parts of lines. The most important is the CHANGE command. The CHANGE command takes two strings as parameters. It searches the line (or range of lines) specified for the first string and changes the first occurrence found to the second string. The two strings are specified using three delimiters as follows:

delimiter string1 delimiter string2 delimiter

Thus, in the file PROG, to change the incorrect variable specified in line 14 from S to STD, specify:

```
:change 14 #s#std#
:       14          WRITE (6,104) MEAN,STD
:
```

Or else:

```
:line 14
:       14          WRITE (6,104) MEAN,S
:change #s#std#
:       14          WRITE (6,104) MEAN,STD
```

By making the second string a null string (no characters in it), it is possible to delete the characters specified by the first string. Thus, to delete the extraneous period in line 9, specify:

```
:line 9
:       9       102  FOR.MAT(' ENTER DATA POINTS')
:change #.##
:       9       103  FORMAT(' ENTER DATA POINTS')
```

Since the second string can be of different length than the first, missing characters can be added to the line by specifying in the first string enough characters to identify where the missing characters are to go, and then make the second string the same as the first plus the missing characters. Thus, going back to line 7, to add a right parenthesis after the 3, specify:

```
:line  7
:       7       101  FORMAT(i3
:change #3#3)#
:       7       101  FORMAT(I3)
```

Watch carefully to make sure the change made was the change intended. It is very easy to not specify enough characters so that the first occurrence of the characters is not those actually changed. Consider line 11:

```
:line   11
:       11      103  FORMAT(6F3.2)
```

The format field width should be 5, not 3. If the change command is:

```
:change #3#5#
:      11      105  FORMAT(6F3.2)
:
```

The statement label is changed instead. The following command puts it back.

```
:change #5#3#
:      11      103  FORMAT(6F3.2)
:
```

Now if one more character (either the one to the left or the right of the "3") is specified, there are enough characters to identify the correct "3".

```
:change #3.#5.#
:      11       103 FORMAT(6F5.2)
:
```

Another way to solve this problem is to restrict the part of the line which the CHANGE command searches. The CHANGE command only searches between two column pointers which are normally set to column 1 and 255. We could use the COLUMN command to set the left column pointer to column 16, make the change, and then set it back:

```
:column 16
:change #3#5#
:      11      103  FORMAT(6F5.2)
:column
```

Other commands are also affected by the column pointer; see the "Edit Mode" description for details.

If all occurrences of the first string are to be changed to the second string throughout the line or lines specified, the "@A" modifier can be used. Thus, to change the iteration variable I to J in the READ statement of line 10, specify:

```
:line  10
:      10      READ (5,103) (DATA(I),I=1,N)
:change@a ;i;j;
:      10      READ (5,103) (DATA(J),I=1,N)
:      10      READ (5,103) (DATA(J),J=1,N)
:
```

Note that every replacement is verified.

If the same change is to be made several times, the strings need not be specified for the second and succeeding times. If the strings are not specified on a CHANGE command, the last specified strings are used. Thus, after changing I to J in the above example, to change I to J in the DO loop of the CALC subroutine, simply specify the command and the line number range:

```
:change@a 21 23
:       21          DO 10 J=1,N
:       22          X = X+DATA(J)
:       23      10  Y = Y+DATA(J)*2
```

The OVERLAY command is often used to add missing items such as statement labels to a line. The OVERLAY command is specified exactly like the REPLACE command, but instead of replacing the line with the string given, it instead overlays the line with the non-blank characters of the string. Thus, to enter the missing statement label "1" on line 4, specify:

```
:line 4
:       4           WRITE (6,100)
:overlay # 1 #
:       4       1   WRITE (6,100)
:
```

Or else use the second form and be prompted for the overlay:

```
:overlay 4
:       WRITE (6,100)
? 1
:       4       1   WRITE (6,100)
:
```

The editor is often used to search for items in a file, as well as to change the file. Two commands, SCAN and MATCH, are used for searching. Both take a string to search as a parameter.

The SCAN command searches for the first occurrence of the specified string anywhere in the line range specified. If no line or line range is specified, it starts at the current line and searches to the end of the file. If it finds the string, it sets the current line to the line where it found the string:

```
:scan /file #mean#
:       3           REAL MEAN
:
```

To find all occurrences of a string, append the "@A" modifier to the command name:

```
:scan@a /file #mean2#
:       20          REAL MEAN,MEAN2
:       25          MEAN2 = Y/N-MEAN**2
:       26          STD = SQRT(MEAN2)
:
```

The MATCH command is the same as the SCAN command except that instead of searching for the string anywhere in the line, it requires the first character to match at the left column pointer position, the second character at the next column position, and so on. In other words, the searching

process is anchored by the left column pointer. Thus, to find all REAL declarations in the file, specify:

```
:column 7
:match@a /file #real#
:       3           REAL MEAN
:      20           REAL MEAN,MEAN2
:
```

Another difference between the MATCH and SCAN commands is that in the comparison process by MATCH, any blank characters in the string specified are considered as "arbitrary" and match any character in the file. Thus, the string " O M" matches both the COMMON and FORMAT statements:

```
:column 7
:match@a /file # o m#
:       2           COMMON DATA,M
:       5      100  FORMAT(' ENTER NUMBER OF DATA POINTS')
:       7      101  FORMAT(I3)
:       9      102  FORMAT(' ENTER DATA POINTS')
:      11      103  FORMAT(6F5.2)
:      15      104  FORMAT(' MEAN=',F8.4,' STD=',F8.4)
:      19           COMMON DATA,N
:
```

In setting the current line in a file in all of the above examples, the LINE command sets the current line to a specific line number in the file. It is also possible to move relative to the current line by means of a command whose "name" is an integer representing the number of lines to move. Thus, the command "1" moves to the next line, "2" moves to the second line past the current line, and so on. By making the number negative we can move backward; "-1" moves to the preceding line. For example:

```
:line  15
:      15      104  FORMAT(' MEAN=',F8.4,' STD=',F8.4)
:1
:      15.25       GO TO 1
:-2
:      14          WRITE (6,104) MEAN,STD
:
```

Finally, to return from the editor to MTS command mode, use the MTS command:

```
:mts
#
```

At this point, we have covered the basic commands for using the editor, and these should be enough for most simple editing. However, only about a dozen of the more than thirty commands available and only two of the modifiers available have been discussed, not to mention the many other features. The "Edit Mode" description should be consulted for the complete description of the editor.

April 1974

April 1974

INTRODUCTION TO DEBUG MODE FOR FORTRAN

The Symbolic Debugging System (SDS) is a conversational facility for testing and debugging programs. This facility was originally provided for assembly language programs, but it has now been extended to include FORTRAN programs. Using SDS, the user may initiate the execution of a program and monitor its performance by displaying or modifying variables at strategic points in the program. This section provides a brief introduction to the debug mode command language for FORTRAN users. A small sample FORTRAN program is given to illustrate the use of SDS. The complete description of SDS is given in the "Debug Mode" section of this volume.

Figure 1 is a sample program to compute the mean and standard deviation of an array of real numbers. The program consists of three sections: the main program MAIN which reads in the data values and prints the final results, the subroutine CALC which computes the desired quantities, and a blank-named COMMON section which contains the data array. In FORTRAN, the main program always has the name MAIN unless it is explicitly specified otherwise during the compilation.

This program is compiled by the FORTRAN G compiler in *FTN using the MTS command

```
$RUN *FTN PAR=SOURCE=MEANPROG,LOAD=MEAN,TEST
```

The source for the program is read from the file MEANPROG and the compiled object module is written into the file MEAN. The TEST parameter must be specified when use of SDS is expected in order to have the FORTRAN compiler produce symbol table records in the object module. These symbol table records are used by SDS and are necessary to enable the user to debug his program symbolically.

The most common method of invoking SDS for debugging this sample program is with the MTS command

```
$DEBUG MEAN
```

The DEBUG command is the same as the MTS RUN command in the manner in which logical I/O units and the parameter field are specified. Here it is assumed that the program uses logical I/O unit 5 for reading the input data and logical I/O unit 6 for printing the output results. For the present purpose of debugging this program interactively, all input test data is entered from the terminal (*SOURCE*) and all output results are printed on the terminal (*SINK*). If the user wishes to assign these units to files, he may specify them on the DEBUG command, for example,

```
$DEBUG MEAN 5=INPUTFILE 6=OUTPUTFILE
```

SDS signals its readiness to accept a command by printing the prefix character "+" in column one. This prefix character precedes all SDS messages and diagnostics.

When the program has been successfully loaded, the message

```
+READY
+
```

is printed, at which point SDS is ready to accept its first debug command.

```
0001                DIMENSION DATA(50)
0002                COMMON DATA,N
0003                REAL MEAN
0004              1 WRITE(6,100)
0005            100 FORMAT(' ENTER NUMBER OF DATA POINTS')
0006                READ(5,101) N
0007            101 FORMAT(I3)
0008                WRITE(6,102)
0009            102 FORMAT(' ENTER DATA POINTS')
0010                READ(5,103) (DATA(I),I=1,N)
0011            103 FORMAT(6F5.2)
0012                CALL CALC(MEAN,STD)
0013                WRITE(6,104) MEAN,STD
0014            104 FORMAT(' MEAN=',F8.4,' STD=',F8.4)
0015                GOTO 1
0016                END


0001                SUBROUTINE CALC(MEAN,STD)
0002                DIMENSION DATA(50)
0003                COMMON DATA,N
0004                REAL MEAN,MEAN2
0005                X = 0.0
0006                Y = 0.0
0007                DO 10 I=1,N
0008                X = X+DATA(I)
0009             10 Y = Y+DATA(I)*2
0010                MEAN = X/N
0011                MEAN2 = Y/N-MEAN**2
0012                STD = SQRT(MEAN2)
0013                RETURN
0014                END
```

Figure 1. Sample Program

Figure 2 gives the sample output from a sequence of commands used to debug the program. Input from the user is given in the lower case and output from SDS and the program is given in the upper case.

```
#debug mean
+READY
+run
  ENTER NUMBER OF DATA POINTS
   2
  ENTER DATA POINTS
   4.0  4.0

  SQRT ARGUMENT NEGATIVE
+CALL TO "MTS"
+READY
+csect calc
+break is#5 is#12
+DONE.
+run
  ENTER NUMBER OF DATA POINTS
   2
  ENTER DATA POINTS
   4.0  4.0
+ "IS#5" *** BREAKPOINT
+READY
+display n
+ "N" IS NOT DEFINED IN THIS SECTION.
+csect *
+DONE.
+display n data(1) data(2)
+ N   'F'   +2
+ DATA(1)   'E'  4.0000000
+ DATA(2)   'E'  4.0000000
+continue
+ "IS#12" *** BREAKPOINT
+READY
+display mean mean2
+ "MEAN" IS MULTIPLY-DEFINED. DEFINITION USED FROM SECTION MAIN
+ MEAN   'E'  0.25000000E+00
+ MEAN2  'E'  -8.0000000
+csect calc
+DONE.
+display mean
+ MEAN   'E'  4.0000000
+modify mean2 '0.0'
+ MEAN2  'E'    WAS -8.0000000      NOW 0.0E+00
+continue
  MEAN=  4.0000 STD=  0.0
  ENTER NUMBER OF DATA POINTS
 $endfile
+USER PROGRAM RETURN
+READY
+stop
#
```

Figure 2. Sample Output

Since most users are incurable optimists when it comes to running a program for the first time, the RUN debug command is given to determine what the program does on the first try. The comments "ENTER NUMBER OF DATA POINTS" and "ENTER DATA POINTS" are produced by the program, and therefore these two lines in the sample output do not start with the "+" prefix character. The program requires as a response an integer N of format I3 giving the number of data points to be used in the program. The input points are read into the array DATA which is of format 6F5.2.

A very simple set of test data is chosen for the first run. The size of the data set is 2 and consists of the points 4.0 and 4.0. This data set, using a simple mental calculation, yields the results of 4.0 for the mean and 0.0 for the standard deviation. In choosing a test data set, it is wise to choose data which give an obvious and simple answer so that most errors in the program are readily apparent.

After the program is run, the comment "SQRT ARGUMENT NEGATIVE" appears, indicating that an erroneous call to the SQRT library subroutine was made in the CALC subroutine. The FORTRAN library has produced the message indicating that the value of the variable MEAN2 was negative. SDS intercepted the FORTRAN library's return to MTS and returned control to debug mode. Whenever any type of abnormal condition occurs during the execution of the program, such as a program interrupt or attention interrupt, SDS steps in and returns control to debug command mode. This also happens when a user's program calls the resident system subroutines SYSTEM, MTS, or ERROR.

At this point, if the user has a serially-reusable program, he may rerun his program and monitor its performance more closely. For a program to be serially reusable, it must be capable of being rerun several times without being reloaded. All locations which contain constant values which are changed by the program must be initialized by the program <u>during execution</u>. For example, a program containing the statements

```
DATA I/3/
K = I
  .
  .
  .
I = 6
```

would not be reusable, since I would not be re-initialized to a value of 3; but a program containing

```
I = 3
K = I
  .
  .
  .
I = 6
```

would be reusable, since I is set to 3 each time the program is used. In general, serially-reusable programs are easier to debug with SDS than are non-serially-reusable programs, since they can be rerun several times

without being reloaded. If the program were not serially-reusable, then the user would have to reload the program again using the $DEBUG command.

As an aid to monitoring the execution of the program, SDS provides the capability of setting breakpoints. When a breakpoint is encountered during execution of the program, execution is stopped, and control is returned to debug mode. The instruction at which the breakpoint is set has not yet been executed when execution is stopped.

The BREAK command may be used to set breakpoints by specifying the statement numbers at which execution is to be stopped. To refer to statement numbers in FORTRAN programs, a prefix must be used to distinguish the type of statement number being given. A "#" must prefix the statement number if it is an external (user-defined) statement number; for example,

```
BREAK #10
```

sets a breakpoint at the user-defined statement number 10. An "IS#" must prefix the statement number if it is an internal (source-listing) statement number; for example,

```
BREAK IS#10
```

sets a breakpoint at the source-listing statement 10. Only those statement numbers which define executable FORTRAN statements may be used. An executable statement is defined as a statement which is from one of the following categories:

(1) Assignment statements
(2) Control statements
(3) I/O statements

All others, such as DIMENSION, REAL, INTEGER, DATA, COMMON, SUBROUTINE, FUNCTION, ENTRY, and FORMAT statements are not executable. Both internal and external statement numbers must be specified without leading zeros.

Since a program may consist of a main program and several subroutines and common sections, there must be a method for determining to which section statement numbers and other symbols refer. This may be done in two ways.

The CSECT command may be used to globally restrict all statement numbers and symbols to a specified section. In the sample output, the command sequence

```
CSECT CALC
BREAK IS#5 IS#12
```

is used to set breakpoints at statements 5 and 12 of the subroutine CALC. If CSECT CALC had not been given, then the first occurrence of IS#5 and IS#12 would be used. In this case, IS#12 would be in the section MAIN and IS#5 would be in the subroutine CALC since IS#5 is a FORMAT statement in MAIN. The command

```
CSECT *
```

may be used to restore the searching of all sections. If the CSECT command has not been given, SDS searches all sections for statement numbers or variable names and uses the first definition encountered.

The @C keyword modifier may be used to locally restrict a symbol to a specified section. The @C modifier applies only to the symbol to which it is appended and overrides any global restrictions set by the CSECT command. In the sample run, the command

```
BREAK IS#5 IS#12@C=CALC
```

also could have been used to set the breakpoints. The modifier @C=CALC restricts IS#12 to the subroutine CALC. @C=CALC is not needed for IS#5 since the only valid definition of IS#5 is in CALC.

The setting of breakpoints at the internal statements 5 and 12 of CALC was chosen so as to allow a closer inspection of the program near the area where the error was indicated. At statement 5, the input data may be examined before any actual calculations are made. At statement 12, the argument to the SQRT call may be examined.

After the breakpoints are set, the program is rerun. When the breakpoint at IS#5 is reached, execution is stopped and the message

```
"IS#5" *** BREAKPOINT
```

is printed. At this point, the user may enter another debug command.

The DISPLAY command may be used to display variable locations in the program. Scalar variables are displayed by giving the variable name; for example,

```
DISPLAY MEAN
```

displays the contents of the variable MEAN converted according to its type and length. In this case, MEAN is a fullword real variable and its value is printed as

```
MEAN  'E'  0.2500000+E00
```

The code E indicates that the variable is real. The codes for FORTRAN variables are:

E	Real (exponential or floating-point)
F	Integer (fixed-point)
L	Logical
M	Complex
X	Hexadecimal
I	Instruction

Array variables are displayed by giving the array name and its subscripts in the same manner as in the FORTRAN program; for example,

```
DISPLAY DATA(1)
```

displays the contents of the first element in the array DATA.

To display a variable which is in a blank-named common section, the @C modifier (or CSECT command) may be used with the name BLANK to specify the section. In the sample program, the array DATA is in the blank-named common section.

```
DISPLAY DATA(1)@C=BLANK
```

could have been used. Simply using

```
DISPLAY DATA(1)
```

would not have worked if the CSECT * command had not been given first. Instead, an error message would be printed indicating that the symbol was undefined.

If all sections are open for searching, and if a symbol is used in more than one section (or subroutine), then SDS displays the first occurrence of that symbol and issues a warning message. In the example,

```
DISPLAY MEAN MEAN2
```

produced a warning message for MEAN since MEAN is defined in both the sections MAIN and CALC.

After the breakpoint at IS#5 has been reached, the next step is to display some of the input data values for the program to determine whether or not everything seems to be in reasonable order. The values of 2 for N and 4.0 for DATA(1) and DATA(2) indicate that the input data was correctly entered.

A CONTINUE command may then be given to resume execution of the program. After the breakpoint at IS#12 is reached, the user can again check the progress of the program. Displaying MEAN and MEAN2, it is discovered that the values are 4.0 and -8.0, respectively. A quick arithmetic check using the appropriate formulas

$$MEAN = (DATA(1)+DATA(2))/N$$

and

$$MEAN2 = (DATA(1)^2+DATA(2)^2)/N-MEAN^2$$

yields the values 4.0 and 0.0, respectively. Hence, the value -8.0 is in error.

Looking back over the sample program, the user can see that this error was introduced in statement 9 of CALC. That statement should read

```
10 Y = Y+DATA(I)**2
```

Since it is not possible to recompile the program in SDS, the best that can be done at this point is to modify MEAN2 to contain the correct value. The MODIFY command may be used to do this. The first parameter for this command gives the name of the variable to be modified. The second parameter gives the value to be used in the modification; the value must be enclosed in primes, for example,

```
MODIFY MEAN2 '0.0'
```

The value for MEAN2 is now modified to 0.0, and execution of the program may be resumed to determine if the remainder of the program seems to be correct. This time, the correct values for the test data are printed by the program.

Instead of entering a second set of test data, the user probably wants to recompile the program to correct the error in CALC. To terminate the program, the user enters a $ENDFILE (or its equivalent). SDS intercepts the termination of the program and returns control to debug mode. The STOP command may be then used to return control to MTS.

The user may use the RESTORE and CLEAN commands to remove breakpoints from the program that were set by the BREAK command. The RESTORE command removes a specified breakpoint. For example,

```
RESTORE IS#12
```

removes the breakpoint set at statement 12 in CALC. The CLEAN command removes all breakpoints that are set in the program.

Multi-dimensioned arrays are specified in the same manner as linear arrays. For example, the third element in the array specified by the FORTRAN source statement

```
DIMENSION ALPHA(10,10)
```

may be displayed by

```
DISPLAY ALPHA(3,1)
```

A sequence of elements of an array may be displayed using the block notation format. For example, to display the first ten elements of ALPHA, the user may specify

```
DISPLAY ALPHA(1,1)...(10,1)
```

The user should note that in FORTRAN programs, arrays are stored in ascending locations with the first subscript increasing the most rapidly and the last subscript the least rapidly.

Arrays may also be displayed using symbolic subscripts. If, in the FORTRAN program, the variables I and J have the values 2 and 3, respectively, then

```
DISPLAY ALPHA(I,J)
```

displays the element ALPHA(2,3).

Arguments to FORTRAN subroutines may be one of two types:

(1) reference by value, or
(2) reference by location.

When an argument is passed as a reference by value, the actual value of the variable is passed by the calling program to the subroutine. Therefore, a copy of that variable is in both the calling program and the subroutine. Scalar arguments are normally passed in this manner. The subroutine uses its copy of the argument for any calculations done. Upon the return of the subroutine to the calling program, the argument is passed back to the calling program and its copy is updated. Therefore, when displaying an argument of this type with SDS, it is important to keep in mind where the variable is located and when it is displayed.

When an argument is passed as a reference by location, only the location of the argument is passed by the calling program to the subroutine. Therefore, only one copy of the argument exists and it is located in the calling program. Array arguments are always passed in this manner. The subroutine uses the array locations in the calling program for its calculations. When displaying an argument of this type, either the variable name from the calling program or the variable name from the subroutine may be used. Both refer to the same argument.

Most debug commands may be given in an abbreviated format. The minimum abbreviations that may be used are underlined.

BREAK	MODIFY
CLEAN	RESTORE
CONTINUE	RUN
CSECT	STOP
DISPLAY	

An automatic error-dumping facility similar to that provided by the MTS $ERRORDUMP command is provided for batch users. In the event of an error condition occurring during the execution of the program, a symbolic dump of the program which includes all variable locations is produced. This facility may be activated for the sample program by the command sequence

```
$SET DEBUG=ON
$SDS SET ERRORDUMP=ON
$RUN MEAN
  2
  4.0  4.0
$ENDFILE
```

Note that the MTS RUN command has been given instead of the DEBUG command. The error-dump facility may be deactivated by the command

```
$SET DEBUG=OFF
```

The symbolic dump produces the variable storage for the sample program in a format similar to the following:

```
DUMP OF SECTION              VA=5004F0

  RA      SYMBOL    TYPE     VALUE                          HEX VALUE

000000   DATA(1)     'E'   4.0000000                       41400000
000004   DATA(2)     'E'   4.0000000                       41400000
000008   DATA(3)     'E'   0.0E+00                         81818181
 ...       ...              ...
0000C4   DATA(50)    'E'   0.0E+00                         81818181
0000C8   N           'F'   +2                             00000002


DUMP OF SECTION MAIN         VA=5002A8

  RA      SYMBOL    TYPE     VALUE                          HEX VALUE

0000B0   I           'F'   +2                             00000002
0000B4   MEAN        'E'   0.0E+00                         81818181
0000B8   STD         'E'   0.0E+00                         81818181


DUMP OF SECTION CALC         VA=5005C0

  RA      SYMBOL    TYPE     VALUE                          HEX VALUE

0000A0   X           'E'   8.0000000                       41800000
0000A4   Y           'E'   16.000000                       42100000
0000A8   I           'F'   +2                             00000002
0000AC   MEAN        'E'   4.0000000                       41400000
0000B0   MEAN2       'E'   -8.0000000                      C1800000
0000B4   STD         'E'   0.0E+00                         81818181
```

April 1974

CONVERSATIONAL USAGE OF MTS

Terminal devices enable the user to interact with a program, monitor its execution, and make decisions at any point of the execution. Terminal devices may be used advantageously to debug a new program. In batch mode, a single error generally terminates a run, while in conversational mode, the user may intervene when an error becomes apparent and immediately correct the error and try again. Thus only one terminal session may be needed to debug the entire program. MTS is historically a terminal-oriented system and therefore many system programs are available to the conversational user which are awkward or impossible to use in batch mode. Communication to MTS in conversational mode, like in batch jobs, is through the MTS command language.

Users may gain access to MTS from a wide range of remote terminals which may be as complex as a remote computer.

The most commonly available terminals at WSU are:

The Selectric Typewriter Terminal[1] is very much like an ordinary electric typewriter. Its character set includes all commonly used text and programming symbols and it can be connected to the computer over great distances via telephone lines. Its output speed is about 14 characters per second and its carriage width is 130 characters.

The Model 33 Teletype is a commercial telecommunications unit which can be connected to the computer via telephone lines. Its output speed is 10 characters per second and its carriage width is 72 characters. It does not provide lower-case alphabetics. It is, however, the most common and the most inexpensive terminal available.

The Model 35 Teletype is functionally almost identical to the Model 33 but is more ruggedly built.

The Westinghouse 1600 Cathode-ray Terminal is functionally equivalent to a teletype terminal. It is noiseless and can be faster. Its main disadvantage is that no "hard copy" of the output is retained because of display on the screen.

The Portacom is functionally equivalent to a Teletype terminal. It's main advantage is its portability. It can be carried like an attache case.

The following two sections describe the physical characteristics of the most commonly used terminal devices, the Teletypes, the Westinghouse 1600,

---

[1]Various companies manufacture this type of terminal. They all, however, have a standard Selectric typewriter keyboard and carriage.

and the Selectric typewriter. If the user is already familiar with terminals, he may skip these sections.

FUNCTIONAL CHARACTERISTICS OF TELETYPES AND ASCII CODE TERMINALS

The Model 33 and 35 Teletypes and the Portacom have a transmission rate of 10 characters per second and have a carriage width of 72 positions. These models do not provide lower-case characters nor several of the special symbols used in programming. Some models have paper tape reading and punching equipment built into them.

The Westinghouse 1600 Cathode-ray Terminal has a transmission rate (on WSU MTS) of 30 characters per second and a screen width of 80 positions. This model behaves exactly like a Model 33 Teletype except for the keys on the extreme left side which manipulate the display and the cursor.

Most keys and controls are common to all Teletypes. The positions and shapes vary considerably, but their functions are identical.

Most Teletypes are capable of both half-duplex and full-duplex modes of operation. In half-duplex mode, the keyboard is connected to the printer so that whenever a key is pressed it is immediately printed, as well as sent to the computer. In full-duplex mode, the keyboard and printer are not connected. A character is printed only when sent from the computer. The switch for HDX-FDX operation is located on the acoustic coupler attached to the back of the Teletype. Although Teletypes are capable of both half duplex and full duplex, WSU only supports half-duplex mode.

The switch labeled "LINE", "OFF", and "LOCAL" on the front of the Teletype turns it and the acoustic coupler on ("LINE") or just the Teletype ("LOCAL") so that it may be used as a typewriter (or paper tape punch). The Westinghouse terminals have two rocker switches -- ON-OFF and LCL-RMT on the far right side to serve the same function.

The ordinary character keys on the keyboard act much like typewriter keys except that as their symbol is printed (if operating in half-duplex mode), the code for that symbol is sent to the computer. The SHIFT key selects the upper character on dual character keys. Note that some keys do not have an upper-shift function and may not be pressed in combination with the SHIFT.

A special key labelled CTRL for 'control' is similar to the SHIFT key in that it selects an alternate function for a key. The code sent to the computer by a key pressed in combination with the CTRL key usually does not represent a printable character and thus no symbol is printed. Some CTRL combinations are used for editing purposes by the device support routines (for example, backspace and 'end-of-line'). These combinations have been designated in this guide by 'CONTROL-x' where x is the character key operated in conjunction with the CTRL key. Note that the labels on the upper part of alphabetic keys generally refer to the control function, not to the shift function. Some Teletypes have color-coded labels: white labels for shift functions and red labels for control functions.

The LINE FEED key causes the paper to move up one line without changing the lateral position of the typing element. It also sends a LINE FEED character to the computer. This key is not used for any special purpose.

The RETURN key causes the typing element to return to the beginning of the line without spacing the paper. It sends a RETURN character to the computer and informs the computer that the user has completed a line.

The RUBOUT key sends a character which is never received by MTS.

The REPT key, when pressed in conjunction with any character key, causes that character to be repeated until the key is released. This is useful for spacing forward or for multiple 'backspaces'.

The BREAK key causes an attention signal to be sent to the computer to interrupt the current operation. This key is located on the right side of the keyboard on the Model 33 Teletypes and to the left on the Model 35. Any input or output in progress will be terminated. Some Teletypes (ones with a built-in telephone dial) have a BRK RLS key, located at the left of the keyboard, which must be pressed after the BREAK key has been pressed, if it lights up.

FUNCTIONAL CHARACTERISTICS OF SELECTRIC TYPEWRITER TERMINALS

When the Selectric typewriter terminal is not connected to MTS it may be used as an ordinary typewriter. Power to the terminal is controlled by an ON-OFF rocker switch on the right side of the keyboard. This switch should be turned OFF when the terminal is not in use.

Most of the normal typewriter controls exist on a Selectric terminal. There is a lever on the right rear of the typewriter cover which reduces the pressure on the paper and allows paper adjustment. A lever on the left adjusts the typing-head angle to compensate for varying thicknesses of paper. The red-topped lever on the carriage may be adjusted to increase the striking force of the typing head for darker copies. The "golf ball" typing elements may be changed to provide a variety of character sets.

Left and right margin stops limit the travel of the typing element. The MAR REL key temporarily releases these stops. The margins may be used in conjunction with the margin editing facility of the device support routines to position the printing on the paper. A small pointer rides in a slot between the margin stops and indicates the current printing element position.

Physical tab stops may be set using the CLR-SET rocker switch on the left side of the keyboard. These may be used in conjunction with the input tabulation editing facility of the device support routines.

The Selectric terminal keyboard is very similar to that of an ordinary electric typewriter. When a character key is pressed, the symbol is printed

on the paper and at the same time the internal code for that character is transmitted to the computer. The shift key allows upper and lower-case alphabetic characters to be produced and selects the upper or lower symbol on dual-character keys.

The TAB key causes the typing element to move to the next tab stop while sending the tab character code to the system. The BACKSPACE key moves the typing element back one position while sending a backspace character code to the system.

The RETURN key causes the typing element to return to the left margin and spaces the paper up one line. It also sends a return character to the system and locks the keyboard. The RETURN code informs the computer that the user has completed a line. The keyboard remains locked until the system again requests input. While locked, all keys except ATTN are inoperative.

The ATTN key is located on the upper right keyboard. When this key is pressed, an attention interrupt is signaled to MTS.

To the left of the keyboard are three lights labeled READY (blue), PROCEED (green), and CHECK (red). The READY light is on when the terminal is connected to the Transmission Control Unit. When this light goes out the terminal has been disconnected from the system. The PROCEED light is on when the system expects input. The keyboard is unlocked until the RETURN (or ATTN) key is pushed. The CHECK light goes on when a character has been incorrectly sent to (or received from) MTS. If the CHECK light is on when the PROCEED light goes on, the CD key must be pressed before the keyboard unlocks. If the CHECK light goes on, it is a good idea to check the line just transmitted to (or received from) MTS for errors.

GENERAL OUTLINE FOR USE OF A TERMINAL

To initiate a conversation with MTS on a terminal device, it generally is first necessary to dial into the system via the telephone lines. There are two types of connections available to MTS through the phone lines. The first, called the ten minute line, is a connection which only allows an elapsed terminal time of ten minutes per signon. If at the end of ten minutes the user has not signed off the ten minute line, MTS will sign the user off automatically. The second line called an unlimited line, does not have a specific elapsed time limit set on the line. The telephone numbers are:

577-0230 for a ten minute line
577-0210 for an unlimited line

Each telephone number listed is the first of an automatic trunk-hunting group. If the first number is busy, the telephone equipment automatically searches through the group of lines until a free line is found. If dialing from a phone which is part of the Wayne State University Centrex System, only the last five digits must be dialed. If there is no answer, dial

(57)7-4799 for a recorded message on the status of MTS. If the line is busy, try again in a few minutes.

Once the connection has been made, type the word "GO" and press RETURN. The system then causes an introductory line to be printed out which identifies MTS and contains the following information:

(L0xx-yyyy)

where "L0xx" is the code for the system device answering the call. Each port is considered a separate device by the system. "L0" identifies the MEMOREX transmission unit, "xx" identifies a particular line of the transmission unit, and "yyyy" is the task number assigned to this session. This information is useful if difficulties arise in the operation of the terminal device. These numbers will aid the Computing Center personnel in tracking down the problem.

If the terminal device is a Teletype, it must identify itself. Some Teletypes are equipped to automatically respond with a unique answerback identification code. If the answerback is automatic, the device prints its identification code, and is then ready to begin the session. Otherwise, the user must manually enter an answerback. See the terminal guides below for details on how to sign on with each type of terminal.

After all the initialization has been taken care of, MTS types the prefix character "#", and is ready to receive the first input line. The first command that should be entered is

$SIGNON ccid

where "ccid" is the user's MTS signon ID. The system then prompts him for his password by typing

```
#ENTER USER PASSWORD.
?
```

The "?" is the prefix character used by MTS to prompt for a response from the user. For each device there is a method to mask the printing of your password. (See the individual terminal guides below.) If the signon ID "ccid" is currently active and is allowed terminal time, and if the password is correct for the given ID, MTS will type out further information for the user on the terminal. This information takes the form:

```
#**LAST SIGNON WAS:  time date
# USER "ccid" SIGNED ON AT time ON date
```

This information gives the time and date for the last time "ccid" was signed on and the time and date for this sign-on. This information may be useful in detecting illegal use of the ID. After the above information has been typed, MTS again types a "#" on the terminal device indicating that the user is now signed on and that MTS is waiting for his next command. The user may now start his "conversation" with MTS, requesting the services he needs and providing the information MTS needs.

Prefix Characters

The first character of each line identifies who is "writing" or who is "reading". On output lines, the prefix character is typed ahead of the message. When input is requested, the prefix character is typed at the beginning of the line and then the system waits for a response. If automatic line numbering is enabled, the prefix is followed by the line number. The prefix character for MTS command mode is the "*". For a complete list of prefix characters, see the "System Command Language" section in this volume.

Conversational Operation

During operation, the terminal device is in one of three modes: receive mode, transmit mode, or idle mode. Since Teletypes provide no special indication of when input is allowed, it is the user's responsibility to be aware of the current mode and act accordingly. On a Selectric terminal, the keyboard is locked whenever input is not allowed, so there is no possibility of typing at the wrong time. A description of each mode and the action to be taken by the user follows.

RECEIVE MODE: The terminal device is in receive mode whenever a message is being transmitted from the system to the device. The first character of the output line is a prefix character that identifies which component of the system produced the output line. The terminal user should not type on the keyboard while the terminal device is in receive mode. However, output from the system may be interrupted at any time. (See "Attention Interrupts" below.)

TRANSMIT MODE: The terminal device is in transmit mode whenever the system is waiting for the terminal user to type a line of input. A prefix character is printed at the first position of the line to show that input is expected and to indicate which component of the system is calling for input.

IDLE MODE: The terminal device is in idle mode when there is no current input or output activity. This situation may occur when a program is being loaded for execution, or when a considerable amount of computing must be done before another input record is required, or before another output line is transmitted. If the terminal device is in idle mode a signal is sent to it once every 28 seconds. This signal does not cause any printing to take place, but it does cause the terminal device to go through some internal antics that produce a little bit of noise. This occasional noise can be taken as a signal that the system is still operating. The user should not type on the keyboard while the terminal device is in idle mode. Entering characters during this time on a Teletype will cause an attention interrupt. (See "Attention Interrupts" below.) On a Selectric terminal the keyboard is locked in idle mode.

Control Characters

Control characters are provided to enable line editing from the terminal device. Each terminal device has its own special characters or combinations of characters defined to act as its control characters. There are five main functions which are provided:

(1) To delete all the characters of the input stream up to the "line-delete" character.
(2) To delete the last character of the input stream. On some terminal devices the carriage actually backspaces and types over the deleted characters when the user corrects errors; on others, such as Model 33 and 35 Teletypes, the carriage does not backspace. On a Westinghouse 1600 the cursor may be backspaced either by the controls on the left of the keyboard or by entering the CONTROL-H combination.
(3) To end an input line and to transmit the entered line to MTS.
(4) To signal an end-of-file.
(5) To signal that the next character should not be interpreted to have any special meaning, but should be transmitted literally.

Table of Control Characters

	Teletype	Selectric
End-of-line Character	RETURN or CONTROL-S	RETURN
Delete-line Character	CONTROL-N	_ (underscore)
Delete-previous Character	CONTROL-H	BACKSPACE
Literal-next Character	CONTROL-Z	!
Attention Interrupt	BREAK	INT or ATTN
End-of-file	CONTROL-C	¢

Input Restrictions

Some terminal devices are able to produce lowercase as well as uppercase characters. Normally alphabetic characters in an input line are converted to uppercase characters before the line is transmitted to MTS. Thus "$SIGNON" and "$signon" produce the same effect. If this conversion is not desired, a device command is available which enables lowercase letters to be transmitted.

Each terminal device has a maximum input line length, but for records read in MTS command mode (prefix character "*"), input may be continued from one line to the next. If the last character of a line is a minus sign "-", then the next line is assumed to be a continuation of the input line. The first character of the next line replaces the "-" of the previous line. There is a restriction on all terminal devices that the total length of an input line may not exceed 120 characters. Up to 255 characters in an input line may be sent to MTS by sending continued lines (lines terminated with a minus sign).

The maximum output line length defaults to the carriage width of the terminal device. However this may be extended up to 255 characters by the "length" device command. Extended output lines are continued as multiple carriage lines until the entire line is printed.

Attention Interrupts

Normally an attention interrupt is a signal to MTS to interrupt whatever it is doing, and to return for another command or data line. An attention interrupt can be used, for instance, to interrupt a LIST command if all the pertinent information has been typed out, or to interrupt a program in execution, if an error is spotted. A program initiated by a $RUN command that has been interrupted (after execution begins) may be restarted at the point of the interruption by using the RESTART command. There are a few occasions when an attention interrupt from the terminal device is ignored. These are:

(1) during the sign-cn procedure,
(2) during the sign-off procedure,
(3) when an attention interrupt has already been received, but not yet processed,
(4) when the executing program refuses to honor the interrupt.

Otherwise, the user may interrupt either during input or output operations. Since each device has its own method for generating an attention interrupt, see the individual terminal device guides below for the procedure. What happens next depends on many things, but generally the comment

```
ATTENTION INTERRUPT AT xxxxxxxx
```

or the comment

```
ATTN!
```

is typed out. The first comment occurs only if a program was in execution at the time of the interruption. In this case, "xxxxxxxx" is the hexadecimal address at which execution was interrupted. The second comment is given if no program was in execution when the attention was received. After one of the above comments has been printed, MTS will type the prefix character "$" to indicate that once again MTS is ready for an input command. At this time the user may enter a new input line. (Note: some system components or user's programs may intercept the attention interrupts, rather than allowing MTS to service the interrupt. For such components, the prefix character printed after the attention has been recognized is the prefix used by that component.)

Error Conditions

A number of error conditions may result during the transmission from the terminal device to MTS. When this happens, an error message is printed out on the terminal device. One of the most common error conditions is the data check which indicates that the characters transmitted by the terminal were not received correctly by MTS. In this case the message printed is:

LINE DELETED: DATA CHECK

and a new prefix character is printed out so the user can re-enter the line. All the error messages take this form, giving the "line deleted" message and the reason for the deletion. If a line greater than 120 characters long (including control characters) is sent, the message:

LINE DELETED: LOST DATA

appears. The line must be re-entered (continued with a "-" if necessary) in lines of less than 120 characters.

Device Support Routines

The MTS system contains a set of subroutines for each type of terminal which interfaces that terminal with the system. These subroutines are called Device Support Routines or DSRs. The DSR for each terminal is responsible for doing the actual I/O with the device, performing error recovery if necessary, recognizing attention interrupts, helping MTS with the sign-on/sign-off procedures, and so on. The DSRs also provide the conversational users with several typographical services which are discussed below.

Device Commands

The terminal user may issue commands to the Device Support Routines. These are not to be confused with MTS commands or data lines. Since the DSRs inspect the input from the terminal first, they can intercept commands intended for them. Such commands are called "device commands" because they deal with device-oriented functions and not system operations. Basically, device commands enable the user to control the formatting of input and output at his terminal. Device commands fall into six groups as follows:

(1) Commands that allow the user to describe the carriage format for his terminal; these include left and right margin settings and tab stops.
(2) Commands that allow the user to specify upper-case conversion and/or hexadecimal input.

(3) Commands that allow the user to redefine the characters having special significance on input (literal-next character, etc.).
(4) The length command which allows the user to establish the truncation length for output lines.
(5) The reset command which reinitializes everything that can be changed by a device ccmmand.
(6) Commands which control special system functions.

The following lists contain only the more commonly used device commands. For a complete list, see MTS Volume 4 Terminals and Tapes.

Device commands may be issued whenever the system is requesting input. The device command is intercepted by the DSR, and, after it is processed, the DSR again prompts the user for the input line expected by MTS. A correctly specified device command is acknowledged while an invalid command is discarded and produces an error message. Device commands must be prefixed with the "device-command character", initially a percent sign ("%"). This is analogous to the "$" for MTS commands although it may not be omitted. This character may itself be redefined by a device command. Note that all lines beginning with the device-command character (initially %) are treated as device commands. If a data line begins with the device-command character it must be preceeded by the literal-next character (!) or the device-command character must be changed with the DCC device command.

Device commands may also be issued by using the $CONTROL command. The format of the device command is the same as that described below, except the command must not be preceded by the device-command character (%). For example:

```
$CONTROL *SINK* LEN=255
```

A short description of device commands follows. Braces indicate that a choice must be made between two parameters, brackets indicate optional fields, and the elipsis (...) indicates repetition of the previous field.

Commands in Group 1:

```
%LMAR
%RMAR
```

The "left margin" and "right margin" commands determine the logical carriage length -- the maximum number of printed characters in an output line. Input lines are unaffected by this command.

```
Form:  %LMAR=nl
       %RMAR=nr
```

where nl and nr are non-negative integers (less than the physical carriage length) representing the positions of the left and right logical margin stops respectively. The difference nr-nl determines the number of characters printed in each output line. The line begins at the left margin on the typewriter and is truncated after nr-nl characters have been transmitted.

(The prefix character is not counted as part of the line.) Note that nr must be greater than nl.

%TABI (for input)
%TABO (for output)

The tab commands turn logical tab mode on or off, redefine the logical tab character, and set the logical tab stops.

FORM: %TAB{I|O}=ON;[char,]dd,...
%TAB{I|O}=OFF

The single character "char" is used as the logical tab character. The default for this character is the "TAB" key on a Selectric terminal and a CONTROL-I on a Teletype or Westinghouse. The quantity "dd,..." is replaced by a sequence of up to nine integers setting the tab positions. Note that the physical tab stops on the terminal (if any) have no relation to the tab command. The "ON" or "OFF" specifies whether logical tab stops are to be enabled or disabled respectively. The "I" or "O" specifies whether editing is to be done to input or output lines. If editing is enabled, each time the logical tab character is encountered in the line being processed, the tab character is replaced by enough blanks to move the column pointer to the next tab position specified in the TAB command. If the current column is already beyond the last tab position, the tab character is left unchanged.

%GOLF

This command sets the internal tables to correspond to different typing elements ("golf-balls"). It is valid only for Selectric typewriter terminals.

Form: %GOLF={963|988}

The numbers 963 and 988 are the numbers stamped on the typing element. The default is 963. All Selectric typewriter terminals on WSU's campus have this typing element.

Commands in Group 2:

%K

The case conversion command specifies the type of alphabetic conversion for input lines from the keyboard. This command is defined only for Selectric typewriter terminals.

Form: %K={LC|UC}

%K=UC causes all alphabetic input from the terminal to be translated into upper case (this is the default). %K=LC causes alphabetic input to be transmitted in the same case as typed.

%HEX

The hex command controls the hexadecimal input editing mode and defines the hexadecimal input delimiter.

Form: %HEX={ON|OFF}[;char]

where ON and OFF enable and disable this mode, and char is any character. The character "char", if supplied, becomes the hexadecimal delimiter. Default mode is OFF, with "char" set to a prime (').

Hexadecimal editing (when ON) occurs after the usual editing for delete line, and so on. When the delimiter is encountered in an input line, the characters following are interpreted as hexadecimal input, two characters per byte, until the delimiter is again encountered. Commas may be used at byte boundaries and are ignored. Hexadecimal input may be used any number of times in a line but the line must end with a normal character. When invalid hexadecimal characters (other than the digits 1-9 and the letters A-F) are encountered, the message LINE DELETED: INVALID HEX INPUT is given.

Commands in Group 3:

The third group of commands are all of the form:

%com=char

where "com" is a three letter command code describing the function to be assigned a new special character and "char" is the character (any character on the keyboard may be used) to be assigned to that function. The default special characters, their names and the three letter command codes are given in the following table.

Name	Default for Selectric	Default for Teletype	Command Code
delete previous character	bs (backspace)	CONTROL-H	DPC
delete line character	_ (underscore)	CONTROL-N	DLC
end of file character	¢	CONTROL-C	EFC
literal next character	!	CONTROL-Z	LNC
device command character	%	%	DCC

When a special function is assigned a new character by means of the appropriate command, the previously assigned character loses its special status and may be used as text. For example:

%DPC=+
%DLC="

make the backspace and underscore ordinary text characters. The sequence AB+C gives AC, and the line 1AB"2CD results in line 2 becoming CD. Care should be taken when special characters are redefined, as one tends to forget and tries to use the old special characters out of habit.

While it is possible in a few cases to have the same character assigned to two different functions at the same time, it is not recommended.

Commands in Group 4:

%LEN

The LEN command defines the physical length of output lines.

Form: %LEN={ddd|OFF}

where ddd is in integer between 1 and 255. If ddd is less than the logical carriage length (see the %LMAR and %RMAR commands), the output line is stopped after column ddd and continued on as many lines as necessary to either complete the line or fulfill the logical carriage length. Continuation lines are identified by an asterisk in column 1, and prefix characters do not count in this process. %LEN=OFF causes a return to the default mode where the physical line length is the same as the logical carriage length.

Commands in Group 5:

%RESET

Form: %RESET

The RESET command resets everything that was changed by a device command to its default value.

Commands in Group 6:

%BDCST

Form: %BDCST={ON|OFF}

This command enables or disables the operator broadcast messages. If %BDCST=OFF is specified, no messages sent by the operator are printed at the terminal. This command is useful for producing uninterrupted output at a terminal. The command should be used with caution, since the user will not receive any messages from the operator (including notices of shutdown). The default is ON.

%WARN

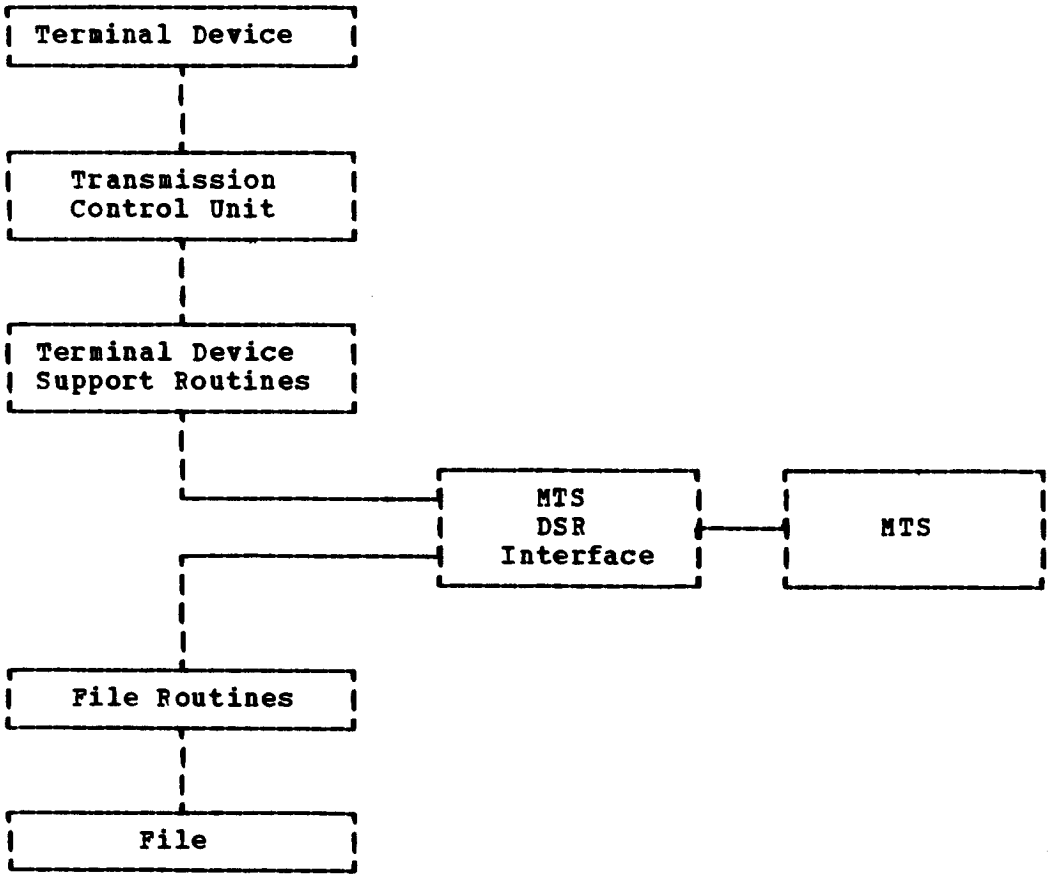
Form: %WARN={ON|OFF}

This command is only defined for ten minute lines (577-0230). If WARN is ON, the message TWO MINUTES LEFT is typed on the terminal when two of the ten minutes remain. This gives time to finish and sign-off. The default is ON.

Processing of Input and Output Lines

For the user sitting at a terminal, there are several levels of processing that an input or output line must pass through. Each of these levels analyzes the line, looking for a particular set of control characters or commands and performs the appropriate conversions. This process may be best explained by using a picture and an example.

Consider the case of a user sitting at a terminal and attempting to enter a line into a file. If he is at a terminal which supports both upper and lowercase letters, he will be concerned about if, and when, lowercase letters are converted to uppercase letters. This conversion may occur at any one of several levels depending on which device commands, global switches, or I/O modifiers have been specified.



After a line has been terminated and transmitted, an interpretation must be made of what has been typed. Line editing is performed either by the transmission control unit or the terminal device support routines, depending on which transmission path is used.

First, the transmission control unit processes an input line by removing rubouts (for a teletype).

Second, the device support routine (DSR) for the MEMOREX unit processes the line editing functions in the following order:

(1) Line termination characters (RETURN)
(2) Literal-next characters
(3) Delete-previous characters
(4) Delete-line characters

If an end-of-file symbol remains as the first character in the input line, a logical end-of-file is returned to MTS; otherwise, the edited line is returned, unless the line is a device command. After a device command is processed by the DSR, a new line is requested from the terminal.

Upper and lowercase conversion may be performed at this level by a DSR. If the device command %K=UC is in effect (the default case), all lowercase letters are converted to upper case. If the device command %K=LC is in effect, all lowercase letters remain as lowercase letters (no conversion takes place).

The third part of the system to process the input line is the DSR interface to MTS. This part processes the I/O modifiers common to all I/O devices. If the user had entered a line using the command

```
$COPY *SOURCE*@UC FILE
```

all lowercase letters coming from *SOURCE* would be converted to upper case. If this uppercase conversion had already been performed earlier, then this modifier would have no effect. Uppercase conversion may also be specified for input lines by giving the MTS command

```
$SET CASE=UC
```

The uppercase conversion specified by the SET command affects only input lines read by the MTS command monitor, that is, those lines entered as MTS commands or data lines when in command mode (with a "#" prefix). Lines entered in all other modes are not affected. Hence, lines entered via the COPY command are not converted by specification of this SET command.

If the user had entered the line using the command

```
$COPY *SOURCE* FILE@UC
```

all lowercase letters coming from *SOURCE* would be converted to upper case by the interface to MTS when the line is being sent to the file routines for processing. Hence the interface to MTS has two chances to process the input line and perform uppercase conversion.

The file support routines are the last part of the system to process the input line. The file support routines perform such functions as indexing

(the @I modifier). After all the line processing is completed, the input line is finally written into the file.

Batch Jobs from a Terminal

It is possible for the user to submit a job via batch from the terminal by using one of the pseudo-device names *BATCH*, *PRINT*, or *PUNCH*. These three pseudo-device names can be used anywhere that an output file or device name (FDname) can appear. (Refer to the section Files and Devices for details.) Thus, for example, one can $GET one of these names, $COPY to it, assign it to SPRINT on a $RUN command, and so on. In the following paragraphs, the term *...* refers to any of the three pseudo-device names.

The actions of these pseudo-device names differ somewhat between batch mode and conversational mode. The following table summarizes the differences.

Pseudo-Device Name	Action in Conversational Mode	Action in Batch Mode
BATCH	A receipt number is printed and the output written to it is used as a batch job.	A receipt number is printed and the output written to it is used as a batch job.
PRINT	A receipt number is printed and the output written to it is printed on the line printer.	The output written to it is written to the line printer.
PUNCH	A receipt number is printed and the output written to it is punched on cards.	The output written to it is punched on cards only if the "CARDS=" parameter is specified on the signon card.

The data written to *BATCH* must contain the $SIGNON command and password, and must contain all the necessary MTS commands to control the job. The data written to *PRINT* are lines which are to be printed on the line printer. No commands are necessary (they would just be printed). The data written to *PUNCH* are lines which are to be punched on cards. No commands are necessary.

In conversational mode (and in batch mode for *BATCH*), when an *...* is opened (that is, when the first line is written on it), a receipt number is printed. When it is closed (when the last reference to it is released), it is released to HASP, and an appropriate message is printed. The next reference to an *...* produces a new receipt number. For example, if a user types:

```
$LIST F *PRINT*
```

the system responds with:

```
*PRINT* ASSIGNED RECEIPT NUMBER 602743
```

soon followed by:

```
*PRINT* 602743 RELEASED
```

Whenever an *...* is opened, certain HASP resources are allocated to process the output, and they remain allocated as long as *...* remains open. However, since the number of HASP resources is fixed, it may happen that all of them are already allocated, in which case the user is told:

```
*...* IS NOT AVAILABLE
```

If this happens, he should wait a few moments and try again.

There are a number of options the user can specify regarding the disposition of an *...*. These may be specified either on the $CONTROL command or by the CONTROL subroutine. Except for HOLD, RELEASE, and CANCEL, these options are the same as the options available on the $SIGNON command for batch jobs. (See the $SIGNON command description for a description of these options.) The following table lists the options and gives the pseudo-device names for which each option is valid. More than one option may appear on a single $CONTROL command. The abbreviation (if any) is underlined.

Option	Pseudo-Device Names
PRINT=TN or PRINT=QN	*PRINT*
<u>R</u>OUTE=xxxx	*PRINT*, *PUNCH*
PROUTE=xxxx	*PRINT*
CROUTE=xxxx	*PUNCH*
COPIES=x	*PRINT*
<u>H</u>OLD	*BATCH*, *PRINT*, *PUNCH*
<u>REL</u>EASE	*BATCH*, *PRINT*, *PUNCH*
<u>N</u>AME='d name'	*PRINT*, *PUNCH*
<u>CAN</u>CEL	*BATCH*, *PRINT*, *PUNCH*

Note that any option not allowed on *BATCH*, may be used on the $SIGNON command of the batch job.

The HOLD option <u>must</u> be used if any other options are desired. For example, simply stating $CONTROL *PRINT* PRINT=TN will <u>not</u> produce the desired result. For any option to take effect, *...* must be open. Specification of the HOLD option for an *...*, opens it and keeps it open until the RELEASE (or CANCEL) option is used. For example, to copy the file named F to the TN printer, the following commands could be used:

```
$CONTROL *PRINT* HOLD PRINT=TN
$COPY F *PRINT*
$RELEASE *PRINT*
```

The HOLD option, since it keeps an *...* open, may also be used to allow copying several files to an *...* and producing a single batch job.

If the *...* is still open, and it is found that an error has been made, the data written to the *...* may be deleted, and the *...* released, by the command:

```
$CONTROL *...* CANCEL
```

If the *...* is already released, the $CANCEL command must be used. If the ATTN or BREAK key is pressed while an *...* is opened, the *...* is automatically held and a message to that effect is printed. The user may then continue, cancel it, or release it.

In the following example, a batch job is set up to compile a FORTRAN program, two documents produced by *FMT are printed as one job on the TN printer, and the output from a WATFOR job is sent to the QN printer. The jobs go immediately into the respective queues (batch and print) and are executed (printed) as soon as possible. The lines typed by the user are in lower case; lines from the system are in upper case.

```
#get -file
#READY
#number
#     1_$$signon myid t=2m p=150 '  batch demo'
#     2_password
#     3_$$create obj size=6p
#     4_$$run *ftn par=source=myfyle load=obj
#     5_$$signoff
#     6_$unnumber
#copy -file *batch*
>*BATCH* ASSIGNED RECEIPT NUMBER 600005
#*BATCH* 600005 RELEASED
#control *print* hold
#*PRINT* ASSIGNED RECEIPT NUMBER 600007
#control *print* print=tn name='w smith'
#run *fmt scards=doc1 sprint=*print*
#EXECUTION BEGINS
#EXECUTION TERMINATED
#run *fmt scards=doc2 sprint=*print*
#EXECUTION BEGINS
#EXECUTION TERMINATED
#release *print*
#*PRINT* 600007 RELEASED
#run *watfor scards=prog sprint=*print*
#EXECUTION BEGINS
#*PRINT* ASSIGNED RECEIPT NUMBER 600011
#EXECUTION TERMINATED
```

```
#*PRINT* 600011 RELEASED
#
```

Since *BATCH* is, in reality, a separate job entry in the execution queue, *BATCH* is not executable if you are still signed on the terminal under the same signon id that you sent to *BATCH*. *PRINT* and *PUNCH* do not operate in the execution queue. Consequently, this rule does not apply to either of these pseudo-devices.

Terminating a Session

When the terminal device is in transmit mode, the user should type in a $SIGNOFF command. After the command line is scanned, MTS closes all files and types out a number of statistics gathered about the use of the computer during the session. These statistics include:

The time of sign-off
The elapsed time in seconds
The CPU time used in seconds
The CPU storage virtual memory integral in page-minutes
The wait-state virtual memory integral in page-hours
The number of tape mounts
The tape drive usage in minutes
The number of drum reads
The approximate total cost of the job
The charge for permanent file storage since the last signoff
The remaining balance of funds for the ID

If the user does not want to wait for the statistics to be typed out, he may enter the command

```
$SIGNOFF SHORT
```

with the result that the sign-off statistics are abbreviated.

If the user only wants to know the approximate cost of his session and his remaining balance, he may enter

```
$SIGNOFF $
```

After the statistics have been written, the telephone line to MTS is automatically disconnected.

USING A TELETYPE OR WESTINGHOUSE 1600

Since almost all Teletypes on campus are Model 33 Teletypes, the following description is limited to that model. If your Teletype is

different, such as a Model 35 or either of the ASR models, see the person who is leasing it or call Academic Services (577-4778) if you need assistance.

Initiation

Most Teletypes and the Westinghouse 1600's are capable of both half-duplex and full-duplex mode of operation. For use with MTS, a Teletype must be capable of half-duplex operation, and must be put into this mode.

Turn the main switch on the Teletype to "LINE", or press the "ON" and "LINE" switches on the Westinghouse. If necessary, also turn on the acoustic coupler. Allow the Westinghouse about 20 seconds for warm-up. Place the telephone receiver in the acoustic coupler in the appropriate position (the position is marked) and dial the MTS number (577-0210 or 577-0230). If, after a few seconds, the light on the coupler fails to come on, lift the receiver and listen. If the line is busy, try again in a few minutes. If there is no answer, call 577-4799 for a recorded message on the status of MTS. When the light on the coupler comes on, type the word "GO" and press the return key. The Teletype then prints the introductory lines:

```
MTS(L0xx-yyyy)
WHO ARE YOU?
```

asking the Teletype to identify itself. Most Teletypes on campus do not have automatic answerback mechanism and require a manual answerback. Since MTS does not make use of the answerback, it is sufficient to press "RETURN".

The initial word "GO" tells the MEMOREX transmission controller that the terminal is a Teletype (or Westinghouse) and automatically sets the speed. On a Teletype, the speed is fixed at 10 characters per second. On a Westinghouse, the speed is set at installation time for 30 characters per second.

Once all the initialization has been taken care of and the "#" prefix character appears, the user may enter his sign-on and password. To protect the password, the Teletype may be temporarily put into full-duplex mode by using the switch on the acoustic coupler. The password will not be typed as it is entered.

Control Characters

The control characters for the Teletypes default to the values listed in the table below. The first four entries may be changed by using the device commands given in the table. The remainder of the entries are not changeable. For the entries such as CONTROL-H, the key labeled CTRL and the specified letter must both be pressed simultaneously. The name in paren-

theses below the code for the character may appear on the top part of the corresponding key (possibly in red).

Default Character	Meaning	How to Change
CONTROL-H (SOH)	The previcus character is deleted.	%DPC=
CONTROL-C (ETX)	A logical end-of-file is presented to the program. Any other contents of the line are not returned.	%EFC=
CONTROL-N	The current input line is deleted. The Teletype returns to transmit mode for the line to be retyped.	%DLC=
CONTROL-Z	The next character typed is treated as an ordinary character, even if it is a CONTROL-H, CONTROL-C, CONTROL-N or a CONTROL-Z. This has no effect on characters below.	%LNC=
RUBOUT	This character is completely ignored by the MEMOREX.	
CONTROL-Q (X-ON)	This character terminates a transmit operation in a normal manner.	
RETURN or CONTROL-S	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 sent, when it causes MTS to see an invalid command.	

Regardless of the other characters (including other control characters) in it, the line must be ended by a return, X-ON, X-OFF, or WRU before MTS receives it. There is no indication that the CONTROL-H has been entered on the Teletype. The replacement character is typed directly next to the one deleted.

Attention Interrupts

To issue an attention interrupt from the Teletype, the BREAK key is used. If the Teletype is a Model 35, the BRK RLS key must be pressed after the

BREAK. Then the attention interrupt will be recognized and a new command may be entered.

Using a Westinghouse 1600

Except for a few minor differences, a Westinghouse 1600 Cathode-Ray terminal acts exactly like a Teletype. These differences are discussed here. It is assumed that the reader is familiar with the rest of the Teletype section.

The Westinghouse 1600 has two white switches to the right of the keyboard which should be turned to "ON" and "LINE" respectively. The acoustic coupler has a separate ON-OFF switch to turn on. Also the HALF DUPLEX-FULL DUPLEX switch should be in the half duplex position.

Both sets of number keys transmit the same characters. To the left are the buttons that control the display. These buttons currently send signals to MTS. The four arrow buttons move the cursor (the "underscore" that indicates the next position to be "typed") in the direction of the arrows. If held down, the motion continues until the button is released. The left arrow button may be used to backspace over characters that have been deleted by sending a CONTROL-H to MTS. The CLR PAGE button erases the display and moves the cursor to the first character position on the top row of the display. The HOME button shifts the cursor to the first column of the top line.

It is possible to suppress the password by changing the coupler switch to FULL DUPLEX, typing the password, and then moving it back. However, it is easier to hold one's hand over the screen, enter the password, press RETURN and then press the CLR PAGE button (thus erasing the screen).

The screen can hold a maximum of 20 lines. When the bottom line has been filled, the display is automatically shifted up one line to make room for the next. The top line is lost in this process.

USING A SELECTRIC TYPEWRITER TERMINAL

Initiation

For use as an MTS terminal, the LCL-COM switch, located on the rear panel of the terminal (if there is one), must be switched to COM, the power switch on the acoustic coupler, and the main power switch on the right side of the keyboard must be "ON" before dialing the telephone number to connect the terminal to MTS. Next, place the receiver on the acoustic coupler and dial the MTS numbers (577-0210 or 577-0230). The blue (ready) and the green (proceed) lights should come on. If they don't, pick up the receiver and listen. If there is no answer, call 577-4799 for a recorded message of the status of MTS. If it is busy, try again in a few minutes. When the lights

on the terminal come on, type the word "go" and press the return button. MTS will then type the introductory heading "MTS (L0xx-yyyy)" and return control so that the user may sign on. To protect the password, the sign-on and password should be entered on two separate lines. MTS prompts the user for the password and then types over the area where the password will be entered. As long as the password and sign-on are correct, the session with MTS can begin.

Control Characters

The control characters for the Selectric terminals are given in the following table. The first four control characters may be changed by the device commands given.

Default Character	Meaning	How to Change
BACKSPACE	The previous character is deleted.	%DPC=
_ (underscore)	Everything typed before the underscore is deleted.	%DLC=
¢	A logical end-of-file return is presented to the program.	%EFC=
!	The next character typed is treated literally.	%LNC=
RETURN	This indicates the line is terminated.	

Special Input Restrictions

The Selectric terminal is capable of transmitting both upper and lowercase letters. Normally, everything is converted to upper case before it is sent to MTS. If this conversion is not desired, lowercase letters may be enabled by using the device command %K=LC.

A physical input line may only contain up to 120 characters, but through continuations, a line of input may be extended up to 255 characters.

Attention Interrupts

An attention interrupt may be generated from the Selectric terminal by using the ATTN or INT key located at the upper right side of the keyboard.

April 1974

BATCH USAGE OF MTS

INTRODUCTION

MTS batch jobs are turned in to the CDPC Production Control desk directly, through a distribution center, or through the remote batch station for processing. In batch mode, the user prepares a card deck which contains all the MTS commands, translator source statements or object decks, and data lines which are needed to accomplish the desired task (or tasks). Once the user has checked his deck for accuracy, he may submit the deck for processing. A receipt is given to the user, the deck is read into the computer and saved in a special file for later execution. The cards are usually returned to the user immediately if submitted to one of the Production Control Desks but may be saved for pick up by (or delivery to) the user. The job is then processed by MTS, together with many other batch jobs. Later, the results of the processing (the job output) may be retrieved at the Production Control desk, at a distribution station, or at a remote batch station, by the user with his receipt. The specific details of submitting and retrieving batch jobs and their output are given in the Facilities and Services of WSU Computing and Data Processing Center for Academic Users (February 1973) and the manual Introduction to MTS @ WSU (January 1974).

ADVANTAGES AND DISADVANTAGES OF BATCH USAGE

Although many users need to have the interactive capability of conversational mode, batch mode is satisfactory and even advantageous in certain situations. There will be few if any terminal users who will not make at least occasional use of batch mode.

Since there are restrictions on the use of line printers, card readers, and card punches in conversational mode, those users wishing to make use of these facilities generally should use batch mode. This can be done by submitting a batch job, creating a batch job from a terminal (see the description of *BATCH* in "Conversational Usage of MTS" in this volume), or by using *PRINT* or *PUNCH* from a terminal.

Batch mode may sometimes be more economical than conversational usage of MTS. Since the charge for a terminal session is based, in part, upon the elapsed real-time, the terminal user will find that he is being charged $3.00 an hour for just sitting and thinking. When running in batch mode, there is no charge for elapsed real-time, but there are charges for card reading, card punching, and line printing (which may approximate the real-time charges for a similar terminal session).

The major disadvantage of batch mode is that the user has no interactive capability with his job. This limits his opportunities for error recovery and making decisions depending on earlier results or conditions. Since there is currently no way for making MTS commands conditional, every command is executed regardless of earlier results. For example, a disadvantage may arise in batch mode when the user wishes to compile and execute in the same job. If an error has occurred during compilation, the object deck may still be executed even though it is erroneous. This effect may or may not be desirable, but the user has little control over it.

DIFFERENCES BETWEEN BATCH USAGE AND CONVERSATIONAL USAGE

The dollar sign command flag "$" is required before all MTS commands in batch mode. This is necessary to avoid interpreting data as command lines and causing irreversible damage to files.

In batch mode, MTS recognizes the following global parameters from the SIGNON command:

Parameter	Meaning	Default
TIME=t	CPU time limit	30 seconds
PAGES=p	Printed page limit	50 pages
CARDS=c	Punched card limit	0 cards
COPIES=n	Number of copies of printed output	1 copy
PRINT={QN|TN}	Printer character set	Any character set
ROUTE=station	Output station for printed and punched output	Input station
PROUTE=station	Output station for printed output	Input station
CROUTE=station	Output station for punched output	Input station (Default is CNTR if the input station has no punch unit.)

If the user does not supply the parameters above for the $SIGNON command, the default values are supplied automatically and his job is held to them. The above global parameters are ignored in conversational mode.

The $DESTROY and $EMPTY commands do not require confirmation in batch mode. Confirmation is required only in conversational mode.

The $ERRORDUMP and $SET ERRORDUMP=ON commands are effective in batch mode allowing the user to produce error dumps for his job. These commands are not effective in conversational mode.

The MTS pseudo-device names have the following defaults in batch and conversational mode:

	Batch Mode	Conversational Mode
SOURCE	Card reader	Terminal
SINK	Line printer	Terminal
PUNCH	Card punch	Card punch
AFD	Active file	Active file
MSOURCE	Card reader	Terminal
MSINK	Line printer	Terminal
PRINT	Line printer	Line printer
BATCH	HASP batch queue	HASP batch queue

Prefix characters are not printed on the user's output in batch mode.

An input line in batch mode is one full card (80 columns)[1]. If the user wishes to continue an MTS command line or a line being entered into the currently active file, a minus sign "-" can be punched in column 80 to indicate that the next card is a continuation of the first card. Terminal input lines are continued by typing a minus sign in the last column of the line. The maximum length for any line entered in MTS command mode is 255 characters.

The standard IBM 1403 Printer has 132 print positions for each line. Since the leading character of an output line to the printer is normally treated as a logical carriage control character, columns 2-133 of the output line are printed columns 1-132 of the printed output. (See the "Carriage Control" section in Volume 3 for a description of carriage control.) Output lines longer than 133 characters are truncated.

The maximum length of an output line to the card punch is 80 characters. Output lines longer than this are truncated.

STRUCTURE OF BATCH INPUT JOBS

The first card in any MTS batch input deck must contain the SIGNON command with a "$" in column 1 and the letters SIG in columns 2-4 or the letters SIGNON in columns 2-7. The SIGNON command identifies the user to the system and supplies certain pieces of information to MTS and to the HASP batch processor which coordinates the execution of batch jobs within the system.

The pieces of information supplied to MTS are the user's signon ID and the global limits for CPU time, pages printed and cards punched. If the global limits are not specified, the default values, which are given above, are imposed. The information supplied to HASP concerns the number of copies of the user's output to be printed and the character set to be used for printing the output, and the designations of any remote batch station locations for sending printed and/or punched output.

---

[1] Lines submitted via *BATCH* may be up to 255 characters long.

The user's password may be specified on the $SIGNON command, but it is recommended that the password be entered left-justified in columns 1-6 of the second card in the input deck. In this manner, the user's password is not printed on his output, thus providing added protection for the security of his password. The password may be changed by the command

```
$SET PW=xxxxxx
```

where "xxxxxx" is the new password. To prevent the printing of the new password when the $SET command is echoed, it is desirable to use the following sequence of MTS commands:

```
$SET ECHO=OFF
$SET PW=xxxxxx ECHO=ON
```

This turns off the echoing of MTS commands on the user's output, changes his password, and turns echoing back on. In this manner, the second $SET command is not printed on the user's output.

Normally the last card in the input deck is the $SIGNOFF command. This terminates the user's job and causes a summary of job statistics to be printed for the user's job. Omission of the $SIGNOFF command is equivalent to placing

```
$ENDFILE
$SIGNOFF
```

at the end of the deck. The job is still terminated and the statistics are printed.

The following is an example of a short batch input deck for compiling and executing a FORTRAN program.

```
$SIGNON X007 T=60 P=100 '  SAMPLE PROGRAM'
JAMES
$SET ECHO=OFF
$SET PW=BOND ECHO=ON
$RUN *FTN PAR=LOAD=-OBJECT
        .
        .
   FORTRAN source program
        .
        .
$ENDFILE
$RUN -OBJECT
        .
        .
      data
        .
        .
$ENDFILE
$SIGNOFF
```

The user with ID X007 and password JAMES is signed on. The global time limit is 60 seconds and the global page limit is 100 pages. A default punched card limit of 0 cards is imposed. The pair of $SET commands changes the password to BOND. The following cards compile and execute his program. The last card terminates his job.

EXPLANATION OF PRINTED OUTPUT

Associated with each batch job's printed output is a header-sheet and a tail-sheet. These sheets are for ease of identifying the end of one user's printed output and the beginning of another user's printed output when separating them off of the printer. Both the header and tail sheets have the user's job receipt number printed on them. In the center of the header-sheet is a block WAYNE STATE monogram. The header-sheet also gives the date and time that the job was processed by MTS, an echo of the user's $SIGNON command, and the user's signon ID, receipt number, and delivery code (if any) in block letters.

The tail-sheet gives a statistical summary of the user's job and the date and time that the job was processed. In the center of the tail-sheet, the non-zero entries from the following list of job statistics are printed.

The user SIGNON ID
The project number
The time of sign-on
The time of sign-off
The elapsed time in minutes
The CPU time used in seconds
The CPU storage virtual memory integral in page-minutes
The wait-state virtual memory integral in page-hours
The number of cards read
The number of lines printed
The number of pages printed
The number of cards punched
The number of tape mounts
The tape drive time used in minutes
The number of drum reads
The approximate total cost of the job
The charge for plotting time
The charge for disk space used since last sign-off
The remaining balance of funds for the ID
The time of the previous sign-on

EXPLANATION OF PUNCHED OUTPUT

Associated with each batch job's punched card output are three identical leading and three trailing separator cards to identify the beginning of the

user's punched output. Columns 1-6 and 75-80 each contain a hexadecimal FF. Between columns 7 and 74 are the user's receipt number and distribution code (if any). These separator cards should be removed before using the deck.

LOCAL AND REMOTE BATCH

Batch jobs submitted and returned at the CDPC or a distribution center are local batch jobs. Those submitted or retrieved elsewhere are remote batch jobs. Batch jobs submitted via *BATCH*, *PRINT*, or *PUNCH* are treated as local batch jobs unless routed to a remote station.

The CDPC has a remote batch station in the basement of the Science Library for processing batch jobs. This station is connected by telephone line to the 360/67 at the CDPC. Batch input from the remote station is sent via telephone to the 360/67 for processing by MTS, and the output is normally returned via telephone to that station for printing. This remote station has card reading facilities and a line printer for producing printed output. Card punching is not available at this station and the number of pages of output may be restricted since only one printer is available. The process of submitting a batch job and retrieving the output at this station is identical to that at the CDPC. The user may also use *BATCH* from a terminal or at this remote batch center to have his job run (or cards punched) at the CDPC. He may then pick up his output at the CDPC Control Desk or have it delivered. The output station code for the basement of the Science Library is SCIL.

HASP

The following information is not necessary for the running of batch jobs. It is presented only for those users wishing to know more about the operation of HASP.

The MTS batch processing is controlled by a program known as HASP (Houston Automatic Spooling Priority System). HASP was originally developed for the IBM Operating System, but has been revised to interface with the MTS system and accept MTS batch jobs.

A batch job in HASP is processed in five phases:

(1) input phase
(2) execution phase
(3) print phase
(4) punch phase
(5) purge phase

At any one instant, several batch jobs are normally in each phase competing for the use of the various hardware and software facilities. During the

input phase, HASP reads in the job deck from the card reader and stores it on a special disk area to await execution. After the job deck has been read in its entirety, an entry is made on a HASP execution queue which indicates the job is now ready for execution in MTS. At the beginning of the execution phase, HASP creates an MTS task specifying the input disk area containing the job deck and specifying the output disk areas (for printed and punched output); a print disk area is always needed by a batch job, but a punch disk area is set up only if the batch job indicates that punched card output is to be produced. When the job execution is terminated, HASP is so informed, the MTS task is destroyed, and the disk area containing the job input deck is released. At this point execution is over but the job output (print and punch) is still on the output disks. After the execution phase, the job is placed in the print queue to initiate the print phase. During the print phase, the job's printed output is produced on a printer under HASP control. After the print phase, the job is placed on a punch queue if it produced punched output. In this case the job enters the punch phase during which its punched output is produced on a card punch. After the punch phase, or after the print phase, if the job produced no punch output, the job enters the purge phase. Here the job is placed on the purge queue and finally purged from the system. During the purge phase, all disk areas used by HASP for the job are released, if they have not already been released. At this time some information about the job is saved so that a user inquiring about his job can be told it is finished. This information is saved for 96 hours after the purge phase is completed.

As mentioned earlier, at any one time there are normally batch jobs in each of the five phases competing for the hardware and software facilities. Hence, at any given instant, HASP must select the next job to be given processing time via some selection criteria. To do this, HASP uses a priority basis to select jobs for processing. This means that jobs are not necessarily processed in the order in which they were read into the system.

Each job is assigned an execution priority when it is entered on the execute queue and a print priority when it is entered on the print queue. These priorities are numbers between 0 and 15 with the higher numbers being the higher priorities.

The execution priority assigned to each job for execution is based on the CPU time limit specified on the $SIGNON command. Currently the priorities from 10 to 15 and 0 and 1 are unassigned. To compute the priorities, the following tables are used:

Execution Table		Print Table	
Time Estimate (seconds)	Priority	Printed Pages	Priority
≤ 5	9		
6-10	8	≤ 20	9
11-30	7	21-50	8
31-180	6	51-300	7
181-300	5	301-1000	6
301-600	4	> 1000	5
601-1800	3		
> 1800	2		

Note that these priorities are subject to change without notice. The command "$INQUIRE PRIORITY" prints the current table.

To compute the execution priority, the priority number is taken directly from the execution table based on the time estimate on the $SIGNON card. For example, if a job estimates a time limit of 200 seconds, the execution priority is 5. Within a group of jobs which have the same priority, jobs are executed in the order in which they are submitted.

To compute the print priority, the priority is taken directly from the print table based on the number of pages actually to be printed (not the page estimate). If 100 pages are to be printed, the print priority would be 7. When a printer is free, the job with the current highest priority is chosen to be printed next. Within a group of jobs with the same priority, the jobs are printed in the order in which they completed execution.

The punched output from a job, which is written on the punch disk while the job is executing, is always produced after the job has been printed. No priorities are associated with punched output. Jobs are punched in the same order in which they finish printing.

Note that HASP is only one of several programs running on the 360/67 concurrently. The time-sharing aspects of the machine apply to HASP as well as to MTS and user programs in MTS. To control the competition between batch jobs and terminal jobs for hardware facilities, there is a maximum number of jobs which HASP allows in each of the processing phases (other than the input phase). These maxima vary during the day and over the month depending on the character of the MTS load level. At all times, the CDPC attempts to maintain a reasonable balance between batch processing and terminal usage in order to provide acceptable service to both user groups.

The maximum number of batch jobs that HASP schedules for execution at once currently depends on a number of factors related to the load on the system. This "system load" is a quantity computed regularly by MTS, and is based on a number of factors, including:

1. The utilization of the central processor.
2. The number of virtual pages in use.

3. The hardware configuration (number of drums and amount of primary storage).

Various "weights" are assigned to the usage of each of the above factors, and a total system load (a number between 0 and a few thousand) is computed. The weights are chosen so that a value of 100 indicates a fairly heavy load (or the system is operating at about 100% of its capacity). Values greater than 100 indicate that the system is being overloaded, hence service will become increasingly degraded.

All numbers in the above tables are subject to change. Such changes are made to ensure satisfactory service to terminal users while running as many batch jobs as possible. In addition, changes may be made to these algorithms to take advantage of more information than is currently used.

Now for the key question: "When will my batch job be done?" It's like this: with the present hardware, HASP can have batch jobs being read in locally, three being printed and ten in execution all at once. Also the remote batch station can be reading in jobs and adding them to the execution queue. Just when any particular job is done depends on how accurate the other users' time and page estimates are, how many terminal users are signed on and what they are doing, how many tape drives are in use, how much the jobs interfere with each other (using the same disk drive, etc.) and about twenty other factors (all interelated in complex ways).

The $INQUIRE command, giving a job receipt number as the parameter, prints the location of the job in the batch process and the number of jobs queued ahead of it. This command gives the same information available to the operators about the status of a job.

April 1974

April 1974

FILES AND DEVICES

Data for a user's commands and programs is accessed from either files or devices. A file is a logical entity -- a set of lines of information. A device is a physical entity such as a magnetic tape unit, a card reader, or a port for a user's terminal to call. The general specification of a file or device is called a File or Device Name in the MTS manuals, and is abbreviated as FDname. The purpose of this section of the manual is to describe the kinds of files and devices available, to give rules for constructing File or Device Names, and to show how they are used.

FILES

A file is an ordered set of zero or more lines. A line is a string of one or more characters (bytes). Both the maximum number of characters in a line and the maximum number of lines in a file depend on the type of file organization, which is discussed below. In the most restrictive case (line files), the maximum line length is 255 characters, and the maximum number of lines is about 10000 for an average file (depending on the length of the lines).

Long-term storage in MTS is organized on the basis of files and hence is referred to as file storage. These files may contain source decks, object decks, data sets, output listings, writeups, etc.

There are two bases of classification of a file:

(1) The availability of the file.

(2) The type of organization.

These two characteristics are fixed at the time the file is created. The first is determined by the form of the file name; the other is specified by a keyword on the $CREATE command.

The Availability of Files

Public files are files containing components of the system, such as language translators and utility programs. Public files may be accessed by anyone, but are protected against modification. Volume 2 of the MTS manual, Public File Descriptions, contains descriptions of the public files available to the user. Public files are also called system files or library

files. At WSU CDPC the concept of a public file has been broadened to include all files maintained at the Computing Center. Consequently, file names prefixed by "CCAP:", "DEMO:", "HELP:", "NEW:", and "OLD:" are considered public files and have entries in Volume 2.

Private files are files belonging to a specific user and may be accessed only by him, unless he gives permission to others for accessing them. There are two types of private files, permanent and temporary. Permanent private files must be explicitly created by the user with the $CREATE command (or by calling the subroutine CREATE). Once created, they exist until the user explicitly destroys them with the $DESTROY command (or by calling the subroutine DESTROY). For the descriptions of these subroutines, see Volume 3, Subroutine and Macro Descriptions.

Temporary files, also called scratch files, are created when their names are first encountered by MTS, and are automatically destroyed when the user signs off. They may be explicitly created (or explicitly destroyed), but must be explicitly created if other than the default characteristics are wanted. The default characteristics are discussed below and are also given in the $CREATE command description in this volume.

Simple File Names

Permanent private file names consist of one to twelve characters. If more than twelve characters are specified, only the first twelve are used. The name can consist of letters (lower-case letters are automatically translated to upper-case), digits, and some special characters. It cannot contain blanks, commas, semi-colons, left-parentheses, at-signs, or plus-signs. It cannot begin with an asterisk.

Examples:
```
FILE
A_LONG_NAME
321
STATP6.S
```

Internally, the name of the file consists of the user's four character signon ID followed by the external twelve character name. In this way, the names of one user's files are always different from those of other users.

If a user wants access to a file belonging to someone else, he must prefix the file name with the owner's signon ID, separating them with a colon ":". Thus if user AAAA wants to read the file NEWS of user BBBB, he must refer to it as

```
BBBB:NEWS
```

This, however, works only if the owner BBBB of the file has permitted it to be read. See the section "Shared Files" below for details.

Temporary (or scratch) file names consist of one to eight characters prefixed by a minus sign "-". This prefix is called the scratch file character (SCRFCHAR), its initial value is a minus sign, and it can be changed by the MTS $SET command or the subroutine CUINFO. The legal characters for a temporary file name are identical to those for a permanent file name.

Examples:
```
-T
-OBJECT
-1
```

Public file names either
1) begin with an asterisk "*", cannot end with an asterisk, and contain a maximum of sixteen characters including the asterisk, or
2) begin with "CCAP:", "DEMO:", "HELP:", "NEW:", or "OLD:" and contain a maximum of twelve characters excluding the "CCAP:", "DEMO:", "HELP:", "NEW:", and "OLD:" prefixes.

Examples:
```
*FTN
*1
*PROJECTACCOUNT
HELP:DIRECTORY
```

DEVICES

A device is a specific discrete item of hardware, such as a card reader, a magnetic tape unit, or a port into the system that a user's terminal can call.

Simple Device Names

Each device has a four-character name and a three or four-character type. Each physical unit has a unique name, and all similar units have the same type. Since the user generally does not know and should not care which specific devices are being used for his job, he uses pseudo-device names. Device names are generally used only by the operators and the system programmers. Since the user occasionally sees device names, a brief mention is made here.

Pseudo-Device Names

In MTS, execution of a program is normally initiated by a $RUN command. This command requests execution of the program and provides MTS with the name of the file or device where the program is located. This procedure is

followed both for system programs (such as translators) and for user programs. The $RUN command must specify the object program to be run and the file or device names to be used by the program. Thus, to run the FORTRAN IV translator, the $RUN command must specify where the source deck is located and where the object deck is to be placed, along with other information.

These specifications in the $RUN command create immediate problems for the batch user in MTS. Since in a batch job, the input batch deck has been read and placed on a special file (unknown to the user) before the job is started, the batch user has no way of providing in the $RUN command a specific name for the file containing the source deck and/or data cards. This problem is circumvented by the specification of pseudo-device names.

Pseudo-device names are synonyms for the actual files or devices used. They begin with an asterisk, end with an asterisk, and have from one to fourteen characters in between. The same characters legal for simple file names are legal here. There are nine pseudo-device names which are automatically predefined for the user. In addition, others may be defined with names of the user's own choosing when mounting removable volumes such as magnetic tape on devices. (See the description of the $MOUNT in this volume, and the "Magnetic Tape User's Guide" in Volume 4 for details.)

The predefined pseudo-device names are as follows:

SOURCE is defined by MTS as the current input (or source) file or device. Initially, the system defines *SOURCE* to be the same as *MSOURCE*. Thus, for batch runs, *SOURCE* is initially defined as the location of the user's input deck, while for terminal usage, *SOURCE* is initially defined as the terminal at which the user is signed on. Therefore, any reference by a batch user to *SOURCE* is equivalent to a reference to the batch input deck. The batch user may thus use the name *SOURCE* in place of the (unknown) name of the input stream.

The user can redefine *SOURCE* by using the MTS $SOURCE command. If *SOURCE* has been redefined by a user, an attention interrupt at a terminal, or an end-of-file on *SOURCE* when attempting to read a command, causes *SOURCE* to be redefined back to *MSOURCE*.

SINK is defined as the current output (or sink) file or device. Initially, the system defines *SINK* to be the same as *MSINK*. Thus, for a batch job, MTS initially defines *SINK* as the printed output stream for that job, while for terminal usage, *SINK* is defined as the user's terminal. (Thus the batch user need not know the specific name of the place being used to collect his printed output, since he may refer to it by the pseudo-device name *SINK*.)

The user can redefine *SINK* using the MTS $SINK command. If *SINK* has been redefined by a user at a terminal, an

attention interrupt causes *SINK* to be redefined back to *MSINK*.

MSOURCE is defined as the master input (or source) file or device, which is the terminal in conversational operation and the file or device from which the batch job is being read in batch operation. *MSOURCE* may not be redefined by the user.

MSINK is defined as the master output (or sink) file or device which is the terminal in conversational operation and the printed output for batch operation. *MSINK* may not be redefined by the user.

AFD is defined as the current active file or device (if any). The active file or device is established by the MTS $GET or $CREATE commands. See the section "Putting Information into Files" below for details.

DUMMY is defined as an infinite wastebasket for output (lines are accepted and they disappear) and an empty file for input (every time a line is requested, an end-of-file condition is returned). *DUMMY* is particularly convenient for specifying that output is to be ignored.

PUNCH is defined slightly differently for batch and terminal usage. For batch jobs, it represents the punched output for the job. Anything written to *PUNCH* is punched on cards, if the user has specified a big enough punch limit for his job. For terminal usage, writing information to *PUNCH* causes a new batch queue entry to be created to punch the output. A receipt number for use in picking up the punched output is printed at the terminal.

PRINT is the same as *MSINK* in batch. From a terminal, writing information to *PRINT* causes a new batch queue entry to be created to print the output. A receipt number for picking up the printed output is printed at the terminal.

BATCH is defined for both batch and terminal usage. Writing information to *BATCH* causes a new batch job to be created to run the input. A receipt number for picking up the batch job's output is printed on *MSINK*.

For a complete description of *PUNCH*, *PRINT*, and *BATCH*, see the section "Batch Jobs from a Terminal" in this volume.

SIMPLE FILE NAMES

A simple FDname is one of the following:

(1) Simple file name
(2) Simple device name
(3) Simple pseudo-device name

All simple FDnames that are not pseudo-device names are normally interpreted as file names, even if they are identical to device names, thus avoiding confusion between a file and a device with the same name.

To specify that an FDname is a device name, it must be prefixed with a ">".[1] For example,

RDR1 refers to file RDR1
>RDR1 refers to card reader 1

This prefix is called the device name character (DEVCHAR), its initial value is a greater-than-sign ">", and it can be changed by the $SET command or the subroutine CUINFO.

To specify that an FDname is only a file name, it must be prefixed with a "#". Continuing the example above,

#RDR1 also refers to file RDR1

This is generally necessary only when the first character of the file name is a flag character. For example,

-T refers to a temporary file with name T
#-T refers to a permanent file with name -T

This prefix is called the file name character (FILECHAR), its initial value is a pound-sign "#", and it can be changed by the $SET command or the subroutine CUINFO.

MODIFIERS

The action of a file or device may be changed by appending to the simple FDname one or more I/O modifiers. Each modifier consists of an at-sign "@" (mnemonic for "attribute"), followed by the modifier name. A modifier name may be preceded by a not-sign "¬" or a minus-sign "-" to reverse its meaning. For example,

---

[1] A user may not ordinarily refer to a device directly. Pseudo-device names must be used instead.

```
$COPY  A *SINK*@¬CC
```

in a batch job causes file A to be copied to *SINK* (that is, printed) with the modifier specifying no carriage control. A detailed description of the modifiers and their meanings is given in Appendix A and in Volume 3.

LINE NUMBER RANGES

The usage of a file or device may be augmented or restricted by appending a line number range specification to the FDname. The form of this specification is

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

where b, e, and i are each line numbers. The b, e, and i are, respectively, the beginning line number, the ending line number, and the line number increment. Any combination of these three items may be omitted. Trailing commas resulting from the omission of items may also be omitted, but leading and internal commas are required. For omitted items, the defaults are

```
b       1
e   99999.999
i       1
```

Note that in the case of a sequential read operation from a line file, the default value of i is not used. If i is omitted, the next line in the file is read; if i is specified as 1, then an increment of 1 is used. If the e is omitted, an end-of-file indication is returned when the end of the file is reached.

The line number range specifies a range of line numbers to be used for input or output. If it is used with a line file, the line numbers are taken from the file. For sequential files or devices, they are generated by the system using the values given for b, e, and i. Thus for line file A,

```
A(10,15)
```

represents all lines in the file with line numbers between 10 and 15 inclusive, and for tape *T*,

```
*T*(1,322)
```

represents 322 records read from or written to *T*, using line numbers 1, 2, ..., 322. Note that on input the record following the one given by e is also read and moved to the user program buffer. Therefore in the above example, the tape would be positioned after the 323ed record not the 322nd. Details on how the line number range affects the usage of an FDname is given below in the section "Types of File Organization".

CONCATENATION

Although the maximum size of any single file is limited (depending on the type of organization of the file), the amount of data that may be referred to by a given FDname is effectively unlimited, because several files or devices may be automatically chained together (concatenated) either explicitly or implicitly. Explicit concatenation means the user explicitly indicates by the FDname how the files or devices are to be chained. Implicit concatenation means that the contents of the file or device indicate how they are to be chained together.

Implicit Concatenation

Implicit concatenation is indicated as follows:

```
$CONTINUE␣WITH  FDname
```

or

```
$CONTINUE␣WITH  FDname  RETURN
```

(where ␣ represents exactly one blank). Whenever such a line is read from any file or device, reading continues with the file or device FDname and the "$CONTINUE WITH" line is not passed as data to the program issuing the read operation. The "$CONTINUE WITH" line is, however, moved to the user program buffer before the new data line is moved in. This may cause problems in some programs if the "$CONTINUE WITH" line is longer than the next data line.

In the first case (without the RETURN), reading continues with the specified FDname, and the lines following the "$CONTINUE WITH" line in the original file or device are ignored. This is analagous to a transfer statement in programming languages. The second case (with the RETURN) is analagous to a subroutine call in programming languages -- after reaching the end of whatever it "continued with", reading resumes with the line following the "$CONTINUE WITH" in the original file or device.

This implicit concatenation action can be disabled. (Otherwise, there would be no way to inspect a file to see if a "$CONTINUE WITH" line is there.) The IC global switch, which has initial value ON, can be set with the $SET command or the subroutine CUINFO. The MTS command

```
$SET IC=OFF
```

turns off implicit concatenation. In addition, the I/O modifier IC overrides the setting of the IC global switch for all references to the FDname to which the modifier is attached. Thus,

```
$LIST  PHYLE@¬IC(LAST)
```

allows the user to see if the last line of PHYLE contains a "$CONTINUE WITH" line.

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 or devices (with optional modifiers and/or line number ranges) in the order desired, connected by plus signs. For example,

```
-T(1,100)+MYFILE
```

specifies the contents of lines 1 through 100 of -T followed by the contents of MYFILE. Note that line 101 of file -T (if there is one) is also moved to the user program buffer, and is then replaced by the first line in MYFILE. In some programs (never in MTS commands like $COPY) this may cause problems if the first line in MYFILE is shorter than line 101 in -T.

If two or more consecutive simple FDnames in an explicit concatenation are identical, all but the first may be omitted. For example,

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

is identical to

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

If a member of an explicit concatenation uses implicit concatenation (that is, contains a "$CONTINUE WITH" line), the entire implicit concatenation is used as only that member of the explicit concatenation. The following members of the explicit concatenation are not ignored. Note that the FDname in the "$CONTINUE WITH" line may itself be an explicit concatenation.

The processing of the next member of an explicit concatentaion is started whenever a return code of 4 (end-of-file or device) is received for a read or write operation (which occurs whenever the physical end is reached or the ending line number is exceeded). Care should be exercised when using indexed operations on explicitly concatenated files. For a sequential write operation, the line number range specifications may be used to control the flow of data into several files or devices. For example,

```
$COPY A B(1,10)+*SINK*(1,10)+B(11)
```

copies from file A, starting at line 1, 10 lines into file B, followed by 10 lines to *SINK*, followed by the remainder of file A into file B, starting at line 11. The ending line number specified in the line number range for a simple FDname (or the last member of an explicit concatenation) is ignored when performing a sequential write operation. Thus, the command

```
$COPY A(1) B
```

is identical to

```
$COPY A(1) B(1,10)
```

To copy only the first 10 lines from file A into file B, the user may give the command

```
$COPY A(1) B(1,10)+*DUMMY*
```

Note that this is not the same as

```
$COPY A(1,10) B
```

since the line number range (1,10) can contain 0 to 10001 lines.

Some examples of explicit concatenation are:

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

Note that an explicit concatenation of a file or device name effectively specifies a new file name for the duration of the read or write operation. That is, the explicit concatenation

```
MYSOURCE+-DATA+*SOURCE*
```

is effectively the name of a file consisting of the contents of the permanent file MYSOURCE, followed by the contents of the temporary file -DATA, followed by the information on the pseudo-device *SOURCE*.

FDNAMES

The term FDname is defined as either

(1) a simple FDname with optional modifiers and/or a line number range, or
(2) an explicit concatenation of two or more simple FDnames, each with optional modifiers and/or line number ranges.

In either case, the files or devices specified by the FDnames may contain "$CONTINUE WITH" lines. In the cases where a restriction must be made to refer to a single file or device, the term FDname is not used.

ERROR PROCESSING

If the FDname for a file or device is incorrect (such as containing an illegal modifier), an error comment is issued when the FDname is entered. If, however, the FDname is correct but the intended usage is not (such as trying to read from a file that does not exist), then the error condition does not occur until the first time an attempt is made to use the file or device.

In either case, when the error condition is noted, error comments are printed as described below, and then

(1) for batch jobs, a return is made to MTS command or debug mode, and cards in the user's deck are skipped until an MTS command ($ in column 1) is found, or
(2) for terminal usage, the line

ENTER REPLACEMENT OR "CANCEL".

is typed. At this point, the user may either enter a replacement FDname (including the modifiers and/or line number ranges), or else enter the six letters CANCEL, in which case MTS returns to MTS command mode.[1]

Error comments concerning simple FDnames are reported using the same form of the FDname that was entered when the FDname was specified, including any modifiers or line number ranges. If the FDname was specified as part of an explicit concatenation, it is preceded by "...+" (if it was not the first member) or followed by "+..." (if it was not the last member). Thus, the comment

"...+A+..." DOES NOT EXIST

means that the file "A" does not exist, that it was specified as part of an explicit concatenation, but it was neither the first nor the last member. If a replacement FDname is entered in response to one of these comments, it replaces the exact FDname typed out in the comment, in the same relative position in any explicit concatenation. Note that a full FDname must be entered, including all line number ranges and modifiers desired.

The three error messages that occur at "first usage" time are as follows:

"FDname" DOES NOT EXIST. means that FDname was interpreted as a file name and no file by that name exists.

"FDname" IS NOT AVAILABLE. means that FDname is an existing file, but it cannot be accessed at the present time (pro-

---

[1]Note that if the replacement is a file named CANCEL, then the user must enter it as "#CANCEL" -- i.e., prefixing it with the "#" file name character.

	bably because it is on a volume that is not available).
"FDname" IS INVALID.	is produced in all other cases. This message is preceded by an earlier message giving the exact error.

As an example, consider the following piece of a terminal session. Input from the user is in lower-case letters; output from the computer is in upper-case. The numbers in brackets at the right are for reference only.

```
#copy  *source* -a@snark                          [1]
#ILLEGAL FDNAME MODIFIER                          [2]
>line one                                         [3]
>"-A@SNARK" IS INVALID.                           [4]
>ENTER REPLACEMENT OR "CANCEL".                   [5]
?cancel                                           [6]
#                                                 [7]
```

In [1], the user enters a copy command with an illegal modifier. The error message is printed immediately [2]. However, the error occurred in the second FDname of the COPY command, which was not referred to until after the first line is read from *SOURCE* in [3]. At that point, the illegal name is first used, and the "first usage" error message is printed [4], and then the user is prompted for replacement [5]. In [6], the user enters CANCEL, which causes return to MTS command mode ("#" prefix character) in [7].

USAGE OF FDNAMES

There are two types of input/output that a program can do. In the more common case, the program may read or write information, without caring where it's coming from or going to. This case is handled with logical I/O units. In the other case, the program reads or writes on a specific file or device, whose name is either built into the program or is obtained as data from the user. This case is handled with FDUB pointers.

Logical I/O Units

When a program is coded, the file or device name to be used for the program's data may be unknown, so that it is impossible to specify it when the program is translated. Even if this information were known, it would be inconvenient to specify it during translation, since this would require retranslation every time the file or device name was changed. Thus, it is desirable to specify the location of the data at execution time rather than at translation time. To do this, a *logical I/O unit* is used. A logical I/O unit is a symbolic name which is used in a program to specify the source of data for input or the destination of output information. A logical I/O unit

does not name a specific file or device; it simply serves as a reference. When a program is executed, it is necessary to first specify, for each logical I/O unit used by the program, the actual file or device to be used. For example, in FORTRAN, the statement

```
READ (5,100) P
```

requests input from logical I/O unit 5. Before this statement can be executed, the user must specify a file or device name to be used whenever logical I/O unit 5 is referenced. Thus, in the $RUN command to execute this program, the user must specify the keyword "5=FDname" to indicate that when "5" is referenced, "FDname" should be used. For example, if the translated program is in the file OBJECT, then when the command

```
$RUN  OBJECT 5=INFYLE
```

causes the program to be executed, the program reads from unit 5 which specifies the file INFYLE.

The names of the logical I/O units are:

```
0 through 19 (integers)
SCARDS
SPRINT
SPUNCH
SERCOM
GUSER
```

For each of these, except 0 through 19 , there is a subroutine of the same name (described in Volume 3) to do input and/or output on this unit. For 0 through 19 , the subroutines READ and WRITE are provided. These subroutines can be called from both assembly language programs and FORTRAN programs. For example, calling the subroutine SCARDS from a program causes a record to be read from the logical I/O unit SCARDS.

Default Values for Logical I/O Units

Since it is convenient to reduce the amount of typing necessary to specify the information required on a $RUN command, some of the logical I/O units have default specifications. These are assignments of pseudo-device names to the logical I/O units if no assignments are provided on the $RUN command.

Unit	Default
SCARDS	*SOURCE*
SPRINT	*SINK*
SPUNCH	*PUNCH*
SERCOM	*MSINK*
GUSER	*MSOURCE*
0-19	none

For example, if on a $RUN command, the logical I/O unit SCARDS is not specified as a particular file or device, then it is by default assigned to the current input *SOURCE*. Since most of the translators in MTS use SCARDS for source program input and SPRINT for compilation listing output, it is often unnecessary to specify these logical I/O units when running the translators, especially in batch mode. For both batch and terminal use, it is usually necessary to specify the logical I/O unit for the translator output of the resulting object deck. Logical I/O units 0 through 19 have no default specifications in MTS. There are, however, default specifications for the equivalent data-set reference numbers within the FORTRAN IV I/O routines during program execution (see the Fortran User's Guide in Volume 6).

FDUB-pointers

It is occasionally necessary for a program to read or write on a specific file or device. The FDname may be built into the program (as for example a fixed file of error messages) or may be input data to the program. The program uses a fullword quantity called a FDUB-pointer to refer to this FDname when doing input or output on the file or device. A FDUB-pointer is the location of a File or Device Usage Block, maintained by MTS to control the usage of that file or device.

A FDUB-pointer is obtained by calling the subroutine GETFD (see Volume 3) giving it the FDname for the file or device. The FDUB-pointer returned by GETFD can then be used in calls to READ or WRITE to do input or output. It can also be used in calls on a number of other subroutines (see table below). When the program is finished with the FDUB-pointer, the FDUB can be released by calling the subroutine FREEFD.

Subroutine	Purpose
GETFD	To get a FDUB-pointer for a given FDname
READ	To perform an input operation
WRITE	To perform an output operation
CHKFDUB	To check an FDUB pointer for validity
CONTROL	To issue control commands to certain types of files and devices
EMPTY	To empty a file
GDINF	To get information about a file or device
GDINFO	To get information about a file or device
GFINF	To get information about a file
NOTE	To return position information for a sequential file
POINT	To position a sequential file
REWIND#	To rewind a file or device
SETAFD	To make a file active
SKIP	To position magnetic tape files and devices
FREEFD	To release a FDUB-pointer

Subroutines using FDUB-pointers

Detailed information about the above subroutines is given in Volume 3. There are several other subroutines (such as CREATE and DESTROY) which take the name as a parameter. These are also described in Volume 3.

TYPES OF FILE ORGANIZATION

The three types of file organization are

LINE	Line file (default)
SEQ	Sequential file
SEQWL	Sequential-with-line-numbers file

The file organization can be specified as the value of the TYPE keyword on the $CREATE command when the file is created. For example,

```
$CREATE  SF TYPE=SEQ
```

creates "SF" as a sequential file. Note that since the default is LINE, all temporary files that are implicitly created (by the first usage of the file) are line files. If a different type is wanted, the temporary file must be explicitly created using the $CREATE command.

The SEQWL type is really a variation of the SEQ type; hence in the following discussion of line files versus sequential files, what is said for sequential files also applies to sequential-with-line-number files.

Line Files

The basic MTS file type is LINE. A line file is an ordered set of zero or more lines. Each line consists of 1 to 255 characters (bytes). Each line has associated with it a unique line number (for that file) which is not part of the line. The lines in the file are numerically ordered. An exact definition of line numbers is given below; for the discussion here, it is a number from -99999.999 to 99999.999 inclusive. Note that while the line number for each line must be stored in the file, the line number is not part of the content of the line.

By using its line number, any line in the file may be directly accessed, either for reading or writing. An input or output operation on a file which explicitly indicates a specific line to be read or written is called an indexed operation. The more common input/output operations on a file specify a line number at which to begin reading or writing, and continue with the "next" line for successive records read or written. These are called sequential operations. These two methods may be intermixed, for example, using an indexed read operation to get to a given position in a file and then reading sequentially from there.

The type of operation used depends on the sequential/indexed modifier bits supplied for each FDname or input/output subroutine call. If, in the call on a subroutine, the modifier word supplied has the indexed bit set, or if @I was appended to the FDname when it was given, then an indexed input/output operation is performed. The default operation is sequential. (See Appendix A for the description of the modifier amalgamation process.)

Indexed Operations

When an indexed read or write operation is done, the line number given in the I/O subroutine call is used to specify the line to be accessed. A return code of 4 (end-of-file) occurs if there is no line in the file with that line number, or if the line number is outside the line number range given with the FDname. (The beginning and ending line numbers, if not specified explicitly in a line number range, default to 1 and 99999.999, respectively.)

Implicit and explicit concatentation with an indexed operation are handled in the following manner:

For an indexed read operation, if the line selected is a "$CONTINUE WITH" line and implicit concatenation is enabled (the default case), then concatenation occurs and the same line number is selected for the new file or device specified by the FDname in the "$CONTINUE WITH" line.

If the file is part of an explicit concatenation (and not the last member) and a condition that would normally produce a return code of 4 occurs, a transfer to the next member in the concatenation occurs instead. Thus, a return code of 4 on an indexed read to a concatenation indicates that none of the members of the concatenation had a line of that number; a successful indexed read on a concatenation selects the specified line from the first member of the concatenation (starting with the current member) with that line number.

Sequential Operations

When a line file is written or read with an indexed operation, the line number is explicitly given. However, for sequential operations on a line file, the question of the beginning and ending line numbers arises. These are specified via the line number range (discussed earlier) which has the form

(b,e,i)

where b, e, and i are each line numbers as described below.

For a read operation, the beginning line number is b (which defaults to 1, if not given) and the ending line number is e (which defaults to 99999.999, if not given). If the increment i is not explicitly given, it is not used. Thus

A

is identical to

A(1,99999.999)

and specifies all lines with numbers between 1 and 99999.999 inclusive. If, however, the increment i is explicitly given, then it is used to select the next line. Thus

```
A(,,1)
```

is identical to

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

and specifies only the lines with integer line numbers from 1 through 99999 inclusive.

For a write operation, the beginning line number is b (which defaults to 1, if not given) and the increment is i (which defaults to 1, if not given). The first line written has line number b, the second b+i, and so on.

Note that b may be greater than e, and that i may be negative. Thus, reading from

```
A(LAST,1,-1)
```

reads the integer-numbered lines of A in reverse order (if LAST is integer-valued).

Mixed Operations

The following discussion applies to mixed sequential and indexed operations on the same logical I/O unit or FDUB-pointer. Accesses to the same file via other logical I/O units or other FDUB-pointers are independent.

An indexed read or write operation operates on the line specified by the line number parameter in the I/O subroutine call, regardless of the previous I/O operation.

If the first operation on a file is sequential, the behavior is as described above: the line is specified by the beginning line number b.

For a sequential operation that is not the first operation, the "next" line is chosen. For a read without an explicit increment, "next" is the next line in the file after the last one read or written. For a read operation with an explicit increment, the "next" line has the line number of the last line read plus the first multiple of the increment for which there is a line. For a write operation with an explicit increment, the "next" line has the line number of the last line written plus the increment.

Line Numbers

Externally, a line number is one of three forms:

(1) ±nnnnn.nnn

where "n" is a decimal digit (0 through 9). The minimum and maximum line numbers are therefore -99999.999 and 99999.999. When writing a line number, leading plus signs and leading zeros, trailing decimal points, and trailing zeros after decimal points may be omitted. Examples of line numbers of this form are:

5 5.1 5.13 5.137 32505.137 -32505.137

(2) LAST

which has the value of the last (algebraically greatest) line number in the file. If the file is empty, the value is zero.

(3) LAST±m

where "±m" is a number of the form "±nnnnn.nnn" as described above. The value of this is the algebraic sum or difference of the two components; thus LAST-1 does not necessarily specify the line number of the next-to-last line, but merely a line number 1 less than that of the last line.

The internal form of a line number is a fullword binary integer whose value is 1000 times the external form. Thus, a line number whose external form is 1 is stored internally as 1000 (decimal) or 000003E8 (hex). The internal form of a line number must be supplied to the input/output subroutines when requesting an indexed operation, and the internal form of the line number of the line that was read is returned after a sequential read (or a sequential write in the case that the RETURNLINE# modifier was specified -- see the Appendix A modifier description).

Sequential Files

A sequential file is an ordered set of zero or more lines. Each line consists of 1 to 32767 characters (bytes). The lines of a sequential file do not have line numbers and cannot be directly referred to by line number from a program or command.

As the name implies, sequential files can only be read or written sequentially. A read operation always gets the next line, starting with the first line (only), while a write operation always adds the new line at the end of the sequential file.

Line number ranges may generally not be specified for a sequential file. For reading, b and e may be specified, but the b must be 1, for example, A(1,99). For writing, line number ranges may not be specified since a sequential write always adds to the end of the file.

Sequential files generally may only be modified by adding lines at the end of the file, and by emptying or destroying the file. By the use of the NOTE and POINT subroutines, it is possible to "remember" positions when reading or writing a sequential file, and then later reposition to begin reading (or writing) at one of these positions. Positioning to a particular record in a sequential file and then issuing a write causes the file to be truncated at the point after the record was written. However, positioning to a particular record in a sequential file and then using a write with the @SP modifier replaces that record with one the same size, and does not otherwise affect the file. Details on the use of the subroutines NOTE and POINT and the @SP modifier on write operations to sequential files are found in Appendix B.

Usage of Sequential Files

Sequential files may be used whenever the file is read or written sequentially. Sequential files, line files, and devices can be intermixed by implicit and explicit concatenation.

An attempt to do an indexed operation on a sequential file, or an attempt to do a sequential operation starting at other than line 1 causes an error message to be generated. This message is controlled by the SEQFCHK global switch. Issuing the commmand

```
$SET  SEQFCHK=OFF
```

disables this check, and the input or output operations are done as though they had been sequential operations starting at line 1. See the section "Putting Information in a File" below.

Sequential files, although more restrictive in usage than line files, can be larger than line files and are more efficient to access. They are, therefore, preferred where only sequential operations are planned.

Magnetic tape users should realize that sequential files are very similar to tape files, and thus most applications which use tapes could use sequential files with slight modifications. Such operations as forward spacing files and the writing of end-of-file marks do not have meaning with sequential files. Files cannot be rewound by writing the three character record "REW" into the file using the @CC modifier. The subroutine REWIND should always be called to rewind a file (it also rewinds tapes). Finally, it should be noted that blocking of records (i.e., packing of short records into fewer, longer records) is not nearly as space-saving on a sequential file as on a tape file since there are no "gaps" between logical records in a sequential file. However, blocking does improve the efficiency of using sequential files, since the input/output overhead is reduced.

Sequential-with-line-numbers Files

The sequential-with-line-numbers file (SEQWL) variant of a sequential file is designed to save a line file together with its line numbers. This file then can be restored later to a line file with the same line numbers as the original file. The line numbers are only retained as part of the file so that the information in the file can be transferred later to a line file. The line numbers cannot be referred to while they are "frozen" in the sequential file.

If LNF is a line file and SWF is a sequential-with-line-numbers file, then (assuming SWF is empty and LNF has no lines with line numbers less than 1)

```
$COPY  LNF SWF@I
```

saves the line file with its line numbers in SWF, and

```
$EMPTY LNF
$COPY  SWF LNF@I
```

restores that information into LNF, with exactly the same line numbers it had originally. Note that the indexed modifier is necessary on the second FDname of both COPY commands to prevent new line numbers from being generated.

The lines in a sequential-with-line-numbers file must have line numbers that are in increasing order. When adding new lines to a file of this type, it is necessary to insure that each line added has a line number greater than the last one saved in that file. If @I is not specified, causing new line numbers to be generated rather than the original ones being used, then LAST+1 can be specified on the file. Thus for example if SWF (as above) is not empty, then

```
$COPY  A SWF(LAST+1)
```

adds the lines from A to the end of SWF.

CREATING FILES

The availability, organization, and size of a file are all established at the time the file is created. Files are created by using the $CREATE command or by calling the CREATE subroutine from a program. The syntax of the $CREATE command is given in the command description. The CREATE subroutine is described in Volume 3. The following discussion covers the specification of the file characteristics when using the $CREATE command.

The $CREATE command has the basic form:

$CREATE filename

which causes a file of this name to be created (unless there already exists a file of the same name, or the user has exceeded his file space allotment). There are several options which can be specified in the form of keyword expressions.

SIZE=n	n is number of average (40 byte) lines
SIZE=nP	n is number of pages (1 page = 4096 bytes)
SIZE=nT	n is number of 2314 tracks (one 2314 track = 7294 bytes)

For most files, this parameter is not needed. The default size for permanent line files is the smallest possible (one 2314 track) which is enough for about 90 average lines. Since files expand automatically when possible up to 16 times, most data sets can fit into a file of default size. (As a matter of interest, about 40% of all user files in the system are one-track files, and about 20% are two-track files.) The following table gives the default sizes:

	Permanent		Temporary	
	Line	Sequential	Line	Sequential
Amount of Space Allocated	1 2314 track	1 2314 track	5 2314 tracks	5 2314 tracks
Pages Charged	2	2	9	9

For larger files, it is necessary to specify an estimated size. When estimating size in terms of pages (or tracks), it should be noted that there is a certain minimum overhead for each line or logical record in the file. These overheads are as follows:

Type	Min. Overhead
LINE	8 bytes/line
SEQ	6 bytes/line
SEQWL	10 bytes/line

Information on the internal structure of files and approximate formulas for the size required for a file is found in Appendix C.

Files are extended automatically when possible by the system. The conditions for automatic expansion are:

(1) The user has not exceeded his disk space allocation.
(2) Space is available on the direct-access volume on which the file was originally created.
(3) The file consists of no more than 16 distinct allocations (called "extents").

If any of the above three conditions is not met, the error comment

FILE "filename": SIZE EXCEEDED

is printed. A line file expands each time by the amount of the initial request or 7 tracks, whichever is less, up to an absolute maximum size of 109 tracks. A sequential file expands each time by an amount equal to the initial request (unless there is not a large enough extent available on that volume, in which case a smaller extent is taken) up to an absolute maximum size which is one disk pack (4000 tracks; 7294 bytes/track).

TYPE=LINE	Line File	(default)
TYPE=SEQ	Sequential File	
TYPE=SEQWL	Sequential-with-line-numbers	File

This specifies the organization of the file being created.

PUTTING INFORMATION INTO A FILE

There are basically two different methods to initially put a set of lines from *SOURCE* into a file. The following descriptions are equally applicable to batch and terminal usage.

The first method to put lines into a file is to establish the file as the active file, either with the $GET command (if it already exists) or with the $CREATE command (if it does not exist), and then turn on automatic line numbering. In the examples below, F is a line file or a sequential-with-line-numbers file.

```
$GET F
$NUMBER
   •
   •  lines to be put in file
   •
$UNNUMBER
$RELEASE
```

If F is a sequential file, then the SEQFCHK global switch must be disabled.

```
$GET F
$SET SEQFCHK=OFF
$NUMBER
   •
```

```
    •  lines to be put in file
    •
$UNNNUMBER
$RELEASE
```

The other method is to use a $COPY command, copying from the source stream *SOURCE* into the file, as for example:

```
$COPY  *SOURCE* F
    •
    •  lines to be put in file
    •
{end-of-file signal}
```

where the end-of-file signal in batch must be a "$ENDFILE" delimiter (or the physical end of the card deck), or from a terminal, it can be either an end-of-file control character for that terminal or a "$ENDFILE".

These two methods are not quite equivalent in behavior. The differences are that in the first case, lines are being read in MTS command mode, and in the second case are being read in copy mode. (See the "System Command Language" section of this manual for a description of the various modes.)

In the copy mode method (the second case), all lines are transmitted into the file as typed except the first line consisting of "$ENDFILE", which is recognized as an end-of-file delimiter and terminates the copy operation. In order to get "$ENDFILE" lines into the file, the command

```
$SET ENDFILE=NEVER
```

should be issued before the $COPY command is issued, causing the "$ENDFILE" lines to be treated only as data lines. This means however, that "$ENDFILE" cannot be used to terminate the copy. If this is being done from a terminal, an end-of-file control character must be used to terminate the copy. But if this is being done in batch, then the COPY command must be the last command in the deck, because all the remaining cards of the deck are copied into the file.

Note that implicit concatenation is normally enabled. If a "$CONTINUE WITH" line is to be copied rather than taking effect, implicit concatenation must be disabled for the copy, as for example:

```
$COPY  *SOURCE*@¬IC F
```

On the other hand, if implicit concatenation is enabled, a line consisting of

```
$CONTINUE WITH *DUMMY*
```

produces an end-of-file condition in both batch and terminal usage and can be used to terminate the copy.

In the MTS command mode method (the first case), there are two differences:

(1) If the last character of an input line (column 80 of the card, if in batch) contains a continuation character (default is "-"), then the following line is appended to the end of the current line (after removing the continuation character). The value of the continuation character (CONTCHAR) can be changed by the $SET command or the subroutine CUINFO.
(2) If a line beginning with a dollar sign is to be put into the file, it must be typed beginning with two dollar signs. The first is removed before it is transmitted to the file. (Lines with one "$" are treated as immediate commands or delimiters.)

MAKING CHANGES TO A FILE

There are two basic methods of making changes to a file: using the MTS commands to make changes to the file on a line-by-line basis, or using the context editor to make changes either on a line basis or on a context basis.

Changes Using MTS Commands

MTS commands can be used to change one line of a line file, a group of lines of a line file, or to empty or destroy an entire file. Sequential files can be changed only by adding to the end of the file, or by emptying or destroying the file. Single lines or groups of lines in a line file can be changed using the same basic two methods that were used to put information in a file. Entering a $GET command for the file makes it the active file. Then entering a line prefixed by the line number causes that line to be entered into the file with that line number. If there already was a line with that number in the file, it is replaced by the new line. Thus,

```
$GET A
121.3XYZ
```

puts XYZ in as line 121.3 of file A. If there already was a line numbered 121.3, it is replaced. Entering a null line (line number immediately followed by an end-of-line character) deletes the line with that number in the file.

When the user is entering lines into the active file in MTS command mode, he enters a line number immediately followed by the contents of the line (assuming automatic line numbering is off). 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. (The first occurrence is the decimal point.)
(3) A "+" or "-" which is not the first input character terminates the number.
(4) A blank terminates the line number.
(5) Any other special character terminates the line number.

If the character terminating the line number is the line-number-separator character (initially a comma), then this character is used only as a separator and is not part of the line number or the line. For example, to put the three characters "123" in as line 6 of a file,

```
6,123
```

would be typed. The line-number-separator character (LNS) may be changed by the $SET command or the subroutine CUINFO.

The $COPY command can be used to copy lines or groups of lines from a file to itself or another file. Thus

```
$COPY  A+B C
```

puts the contents of A, followed by the contents of B, into file C (or if C is sequential, adds to the end of C). If either A or B are line files, then the "contents" are only the lines with numbers between 1 and 99999.999, which is not necessarily the entire contents of the file. If C is a line file, lines that are not replaced by the copy operation are not affected.

```
$COPY  A(1,5)+(99,99)+(1000) B
```

copies from A lines 1 through 5, 99, and 1000 through the end of A into B, starting at line 1 in B, if B is a line file, or adding to the end of B, if B is sequential. File A must be a line file in this example. Note that if the example were

```
$COPY  A(1,5) B
```

then A could be either a line file or a sequential file.

```
$COPY  A(10,15) A(100)
```

makes a copy in A of the lines in the range of 10 through 15, starting at line 100. Remember that the range of 10 through 15 is the same as 10.000 through 15.000 and can contain anywhere from 0 to 5001 lines. The increment between the copied lines starting at line 100 is 1 (the default).

A copy operation from a file to itself specifying overlapping line number ranges can be done, but should be used with care.

```
$COPY  A(1,9) A(2)
```

results in line 1 being repeated in lines 2 through 10, since line 1 is copied into line 2, and then line 2 (which is now a copy of line 1) is copied into line 3, and so forth.

Once a file has been created, it may be emptied or destroyed at any time. This applies both to permanent files and to temporary files. Permanent files exist until the user destroys them; temporary files exist until the user destroys them or until he issues a $SIGNOFF command (or its equivalent), whichever comes first.

To empty a file, the command

$EMPTY filename

should be issued. This causes the contents of the file to be discarded, but all storage space allocations for the file are preserved. Future references to the file reuse the file space. If the command is issued from a terminal, the user is asked to confirm that he really wants to get rid of the contents of the file. The message is

FILE "filename" IS TO BE EMPTIED. PLEASE CONFIRM.

The user should respond with "OK", "O.K.", or "!" to empty the file. Any other response cancels the command. Confirmation is not requested for temporary files, nor for any files in batch runs.

It should be noted that the $EMPTY command empties the whole file; parts of a file cannot be emptied by attaching a line number range to the file name in the command. Thus

$EMPTY A(10,50)

is identical to

$EMPTY A

and empties all of file A. To partially empty a file, the parts to be saved should be copied to another file, the file should be emptied, and the saved parts should be copied back to the original file.

To destroy a file, the command

$DESTROY filename

should be issued. This causes all space allocated for this file to be released. If this command is issued from a terminal, confirmation is requested:

FILE "filename" IS TO BE DESTROYED. PLEASE CONFIRM.

The user should respond with "OK", "O.K.", or "!" to destroy the file; any other response cancels the command. Confirmation is not requested for temporary files, nor for any files in batch runs.

In conversational mode, no confirmation request is issued when a file is to be emptied or destroyed if the confirmation is provided on the command line. For example, the sequence issued in conversational mode:

```
$EMPTY filename1 OK
$DESTROY filename2 OK
```

empties "filename1" and destroys "filename2" without requesting confirmation.

Changes Using the Context Editor

The context editor can be used to edit line files , by lines or groups of lines. It can also be used on a context basis, scanning for lines with certain character strings in them, replacing characters in a line with other characters, and so on. For example, on a line-number basis, the following replaces line 2, deletes line 7, and inserts two lines after line 8 of the file being edited (the first character is the prefix character; input from the user is in lower-case, editor output is in upper-case):

```
                    :replace 2 'new second line'
                    :      2     NEW SECOND LINE
                    :delete   7
                    :insert   8
                    ?eighth and a quarter
                    ?eighth and a half
null line input->   ?
                    :
```

On a context basis, the following example scans through the file looking for the characters "ABC#D":

```
:scan /file 'abc#d'
```

and then the following command changes "C#D" to "XYZ":

```
:change 'c#d'xyz'
```

The editor is invoked in MTS command mode with the $EDIT command, and has its own set of sub-commands. Full details on using the context editor are given in the "Edit Mode" section of the "System Command Language" section in this volume.

The context editor can only be used on line files at present.

Restoring the Contents of a File

Often a user makes changes to a file and then wishes he hadn't. The user should realize that the context editor makes changes to a file as the commands are entered; it does not make changes to a working copy. Thus, it is prudent to either make a copy of the file as a backup before starting to

edit it, or else to use the editor's checkpoint/restore facility. The CHECKPOINT edit command tells the editor to remember the contents of the file at the time the command was entered. The RESTORE command then allows the user to restore all or parts of the file to the state it was when checkpointed. See the "Edit Mode" section of the "System Command Language" for details.

Recovery of a file that has been accidentally changed on a large scale (including its accidental emptying or destruction) can often be rectified by having the file restored to the condition it was in when files were saved on the previous night. The CDPC does a "file-save" at about three to five a.m. on Monday thru Thursday and on Saturday. A file can be restored to its state as of the file-save for any night of the preceding week, or its state as of the Saturday file-save for any of the preceding three weeks. There is a five dollar charge for restoring a file. Details on how to have a file restored are given in the "File-Restoring Service" section of Facilities and Services of WSU Computing and Data Processing Center for Academic Users.

DISCOVERING THE CHANGES TO A FILE

Often a user, after extensively working on a file, ends up with a "current" copy and an "original" copy, and would like to know how they differ. In particular, he would like a list of changes that must be applied to the original copy to make it into a current copy.

There are three programs available in MTS public files that produce "changes" in this manner: *DOWNDATE, *UNEDIT, and *AMENDS. All three differ on either the basis for deciding if something is changed, or the method of presenting the changes, or both. The rest of this section presents examples showing how they differ. Details on use are found in Volume 2.

The program in *DOWNDATE produces *UPDATE control cards as output. The current and original copies are read sequentially, and may be magnetic tape files, sequential direct-access files, or line files. All line numbers are ignored. The two copies are assumed to contain 80 character logical records. Only the first 72 columns of each record are compared; columns 73 through 80 are assumed to contain sequence-ID's. For example, if the original and current copies are as follows (assuming the sequence-ID begins in column 73):

```
original:     FIRST                                    SEQID001
              SECOND                                   SEQID002
              THIRD                                    SEQID003
              FOURTH                                   SEQID004
              FIFTH                                    SEQID005
              SIXTH                                    SEQID006
              SEVENTH                                  SEQID007
              EIGHTH                                   SEQID008
              NINTH                                    SEQID009
              TENTH                                    SEQID010
```

```
current:    FIRST
            NEW SECOND
            THIRD
            FOURTH
            FIFTH
            SIXTH
            EIGHTH
            EIGHTH AND A QUARTER
            EIGHTH AND A HALF
            NINTH
            TENTH
```

Then

```
            $RUN *DOWNDATE SPUNCH=CHANGES
```

and specifying "original" when it asks for "old source" and "current" when it asks for "new source" yields as content of CHANGES:

```
            %BEFORE 'SEQID002'
            NEW SECOND
            %DELETE 'SEQID002'
            %BEFORE 'SEQID007'
            EIGHTH
            EIGHTH AND A QUARTER
            EIGHTH AND A HALF
            %DELETE 'SEQID007'  'SEQID008'
            %BEFORE FILEMARK
```

which is correct, although not a minimum set of UPDATE commands. Specifying:

```
            $RUN *UPDATE
            %INPUT original
            %OUTPUT new
            $CONTINUE WITH CHANGES
```

makes new into a "current" copy.

The program in *UNEDIT uses the same comparison algorithm as DOWNDATE but puts out context editor control cards. Both the current and the original copies must be in line files. The entire contents of the lines are compared. The line numbers are used for the edit control card output, but are ignored in the comparing process. Thus, the effect of applying the output of UNEDIT as commands to the context editor working on the original produces, as a result, a file with lines that have the same content as the "current" copy and in the same order, but not necessarily with the same line numbers. Given the following example, with "original" and "current" similar as before:

```
          line number    contents

original:      1         FIRST
               2         SECOND
```

```
             3        THIRD
             4        FOURTH
             5        FIFTH
             6        SIXTH
             7        SEVENTH
             8        EIGHTH
             9        NINTH
            10        TENTH


current:     1        FIRST
             2        NEW SECOND
             3        THIRD
             4        FOURTH
             5        FIFTH
             6        SIXTH
             8        EIGHTH
             8.25     EIGHTH AND A QUARTER
             8.5      EIGHTH AND A HALF
             9        NINTH
            10        TENTH
```

Then

```
$RUN *UNEDIT 0=original 1=current  SPUNCH=CHANGES
```

yields as contents of CHANGES:

```
DELETE          2
INSERT          1
NEW SECOND
$ENDFILE
DELETE          7
INSERT          6
EIGTHTH
EIGHTH AND A HALF
$ENDFILE
```

so that

```
$SET ENDFILE=ON
$EDIT original
$CONTINUE WITH CHANGES
```

produces a "current" copy.

The program in *AMENDS considers the line numbers as well as the content in comparing lines. Its output consists of MTS command mode line insertions and deletions. Thus, using the same "original" and "current" files as in the UNEDIT example above,

```
$RUN *AMENDS  0=original     1=current SPUNCH=CHANGES
```

results in the following content of CHANGES:

```
$NUMBER        2     1
NEW SECOND
$UNNUMBER
7
$NUMBER        8.25  .25
EIGHTH AND A QUARTER
EIGHTH AND A HALF
```

so that

```
$GET original
$SOURCE CHANGES
```

results in "original" being converted into "current"

SHARED FILES

[The current facilities for sharing files between users are a minimal interim version. This section will be replaced when the full sharing capability is installed in the system.]

When a file is first created, the owner (the signon id under which the file was created) has unlimited access to it and all other users have no access to it. However, if the owner wishes, he can share the file, making various levels of access available to various categories of users. Shared files are referenced by prefixing the file name with the owner's signon id and a colon. For example, ABCD:XYZ is the file XYZ belonging to user ABCD. Sharing a file requires the use of the $PERMIT command. In order to specify a "permission", a valid access type must be stated. The access type tells who is allowed access to the file and what type of access is allowed.

The six possible parameters of the $PERMIT command are "ALL", "CNW", "NONE", "PRJNO", "RO", and "RUN". The "ALL" access type allows other users to read the file in any way. It may be listed, copied, run, loaded, and so on. To specify the "ALL" access, the command

```
$PERMIT filename ALL
```

should be used. The user is still allowed to alter the file when the access code is "ALL". Problems may arise when another user is reading the owner's file at the same time the owner empties the file. "cannot write" access ("CNW") prevents the owner from writing into the file. (Under no circumstances can one user write into another user's file.) To specify the "cannot write" access, the command

```
$PERMIT filename CNW
```

should be used. "CNW" access is useful in that it prevents the owner from accidentally altering or emptying a file (although it may still be destroyed). Other users are not allowed to read the file by this access type alone. This is the only access allowed for temporary files. "PRJNO" like "ALL" specifies which users are allowed access to the file. When a file is given "PRJNO" ("project") access, only those users with the same project number are allowed access. To specify "project" access, the command

```
$PERMIT filename PRJNO
```

should be used. In order for another user with the same project number to access the file, that user must first copy the file using the *COPY program. (See MTS VOLUME 2, Public File Descriptions for details.) user XXXX owns file HISFILE with PRJNO access type and another user with an identical project number wants to copy the file to -MYFILECOPY, the appropriate command would be:

```
$RUN *COPY SPUNCH=-MYFILECOPY PAR=XXXX:HISFILE
```

To allow a file to be run, or loaded, but not copied, listed, or read by any program other than the dynamic loader, the command

```
$PERMIT filename RUN
```

should be used.

The last two parameters are "RO" (read-only) and "NONE". Specifying "RO" has the same effect as CNW and ALL together, that is, it allows all users to read the file and allows no one, including the owner, to write into it. Specifying "NONE" removes all access information allowing unlimited access by the owner and no access by anyone else (the same as when the file was created). If no access type is given on the $PERMIT command, "NONE" is assumed.

If no parameters are given on the $PERMIT command, the user is prompted for filenames and access types, one set per line, until a null line or end-of-file is given. For example:

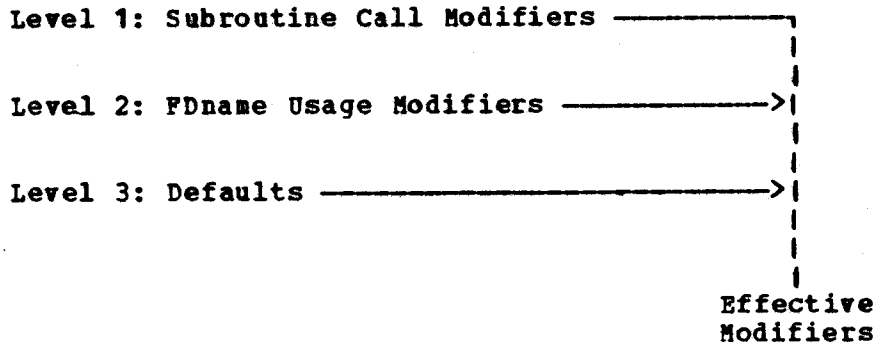
```
$PERMIT
  ENTER FILE NAME AND ACCESS METHOD
MYFILEA ALL
MYFILEB NONE
$ENDFILE
```

APPENDIX A: I/O MODIFIERS

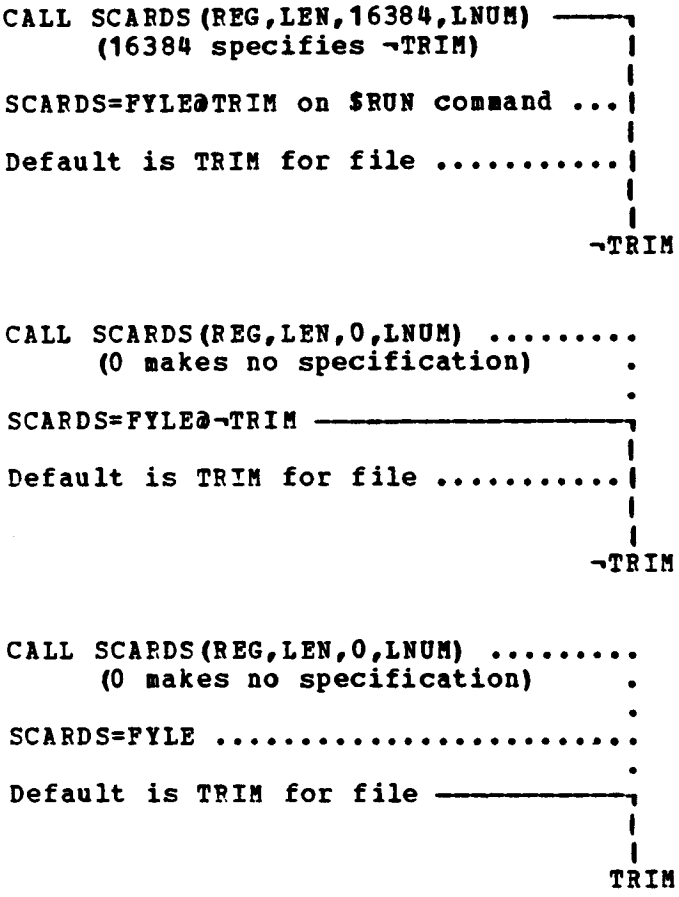
INTRODUCTION

Modifiers are used to modify the action of a specific I/O call or a general I/O usage. Modifiers may be used in I/O subroutine calls (SCARDS, SPRINT, READ, and so on), in macro calls setting up the corresponding I/O subroutine calls, in calls to GETFD, or as parts of FDnames given in MTS commands. Modifiers control such functions as upper or lower-case conversion, logical carriage control, machine carriage control, record trimming, and so on.

In general, there are three levels of precedence in the usage of modifiers. In the first level of precedence are the modifiers specified on a call to one of the I/O subroutines. If the modifier is not specified by the subroutine call, or if a user generated subroutine call is not relevant (e.g., when a $COPY command is issued), the second level of precedence, which consists of the modifier name as part of the FDname, applies. Note that the group of modifiers which can only control the action of a specific I/O call (for example ERRRTN and NOTIFY) are not valid at this level of precedence. If the action of the modifier is not specified by the second level, the third level of precedence, which consists of the default specifications, applies. The default specification depends upon the type of FDname referenced in the I/O call and the settings of global switches. These defaults are given in the explanation of modifier bits below. Modifier specifications given at the first level of precedence override specifications given at the second and third levels. Modifier specifications given at the second level override specifications given at the third level. This precedence process is illustrated in the diagram below. Each modifier pair is treated independently in the above precedence process.



The example below illustrates the three levels for controlling the TRIM modifier.



The action of the modifiers specified on a subroutine call is controlled by a fullword of modifier bits given as one of the parameters to the subroutine. The action of the modifiers on the subroutine call applies only to that specific call. There are two classes of modifiers.

(1) Bits 0-7 are referenced individually, and specify the options for a specific I/O call. If the bit is set, the modifier's action is enabled. If the bit is not set, the default specification is used (which normally means the modifier action is disabled).

(2) Bits 8-31 are referenced in pairs and specify options for general I/O usage. For each option, one bit is used as an "ON" bit and the other as an "OFF" bit. If either of the bits, but not both, is set, the modifier action is as specified. If neither or both of the bits is set, indicating a "don't care" condition at this level of precedence, the modifier name appended to the FDname is used. If there is no

modifier name appended to the FDname, the default specification for the FDname type is used. The normal programming practice is to leave the modifier bits zero on the subroutine call and apply the modifier names to the FDname referenced unless the program depends upon the modifier bits being set for a specific subroutine call. Here is an example done first in assembly language and then in FORTRAN:

```
          CALL SCARDS,(REG,LEN,MOD,LNUM)
            .
            .
REG     DS   20F
LEN     DS   H
MOD     DC   X'00004000'   Specifies no trimming of input lines
LNUM    DS   F
```

Note that if the subroutine call is set up by a macro call, the modifier names rather than the bits are used in the macro parameter list. Thus the above example would become

```
SCARDS  REG,LEN,@¬TRIM,LNUM
```

Here is the FORTRAN version that specifies no trimming:

```
      INTEGER*2 LEN
      DATA MOD/Z00004000/
      CALL SCARDS(REG,LEN,MOD,LNUM)
```

The action of modifiers applied to the FDnames is controlled by the modifier name (preceded by "@") appended to the FDname. The action of the modifiers appended to the FDname applies to all I/O calls referring to that usage of the file or device. If the modifier name is preceded with "¬" or "-", the other bit of the bit pair is set, which negates the action of the modifier name. (The modifier applies only to the FDname to which it is attached.) If implicit or explicit concatentation to another FDname occurs, the modifiers must be applied to both FDnames even if the FDnames are the same. If the user at a terminal is prompted for an FDname, the full FDname including the modifiers and line number range must be given with each request. The order of modifier names appended to an FDname is unimportant. Some examples are

```
FILE1@I@UC                    Specifies indexed and upper case
FILE2@¬TRIM                   Specifies no trimming
*SINK*@NOCC                   Specifies no logical carriage control
*SINK*@¬CC                    Specifies no logical carriage control
RDR1@BIN                      Specifies no EBCD conversion
FILE3@MCC(1,10)               Specifies machine carriage control
FILE3@MCC(1,10)+(20,30)@MCC   Specifies machine carriage control
                                for lines 1 to 10 and 20 to 30
```

If the modifier action is also specified on a subroutine call, the modifier action applied to the FDname is overridden.

April 1974

EXPLANATION OF MODIFIERS

The device types discussed in the exceptions to the default modifier bit specifications are:

PTR	Printers
TTY	Teletype or ASCII code terminals
2741	Selectic typewriter based terminals
2260	IBM 2260 Display Unit
9TP	9-track magnetic tape drive
SDA	Synchronous data adapter (Remote batch via MEMOREX Controller)
HPTR	Printed output under HASP

The values indicated below with each bit specification are the values that the modifier word for a subroutine call would have if only that modifier option was specified.

Bit 31	SEQUENTIAL, S	Value:	1 (dec)	00000001 (hex)
30	INDEXED, I		2	00000002

Default: SEQUENTIAL
Exceptions: None

In general, the INDEXED modifier is applied only to line files (or sequential-with-line-number-files), while the SEQUENTIAL modifier is applied to line files, sequential files, and all types of devices. Note that the SEQUENTIAL modifier and the sequential file are not directly related. The paragraphs below describe the action of this modifier pair and the results that occur when these modifiers are not used in the normal manner.

With each logical unit (or FDUB-pointer), there is a current line pointer which contains the line number of the last record read or written. When an I/O operation is performed, the current line pointer is first set to the line number of the record to be read or written before the actual read or write occurs. After the read or write operation has occurred, the current line pointer contains the line number of the record last read or written.

I/O operations involving line files (or sequential-with-line-number files) may be done with either SEQUENTIAL or INDEXED specified. A SEQUENTIAL I/O operation occurs when the user specifies that the "next" record is to be read or written. For a read operation, "next" means the record that is next in ascending line number order from the current value of the line pointer (last line read or written) of the same logical I/O unit (or FDUB-pointer). If, however, an increment was explicitly given with the FDname, the line number read is the current value of the line pointer plus the first multiple of

the specified increment for which there is a line in the file. For a write operation, "next" means the current value of the line pointer (last line read or written) plus the increment specified with the FDname (defaults to 1) of the same logical I/O unit (or FDUB-pointer). An INDEXED I/O operation occurs when the user specifies the line number of the record to be read or written. As an example, consider the following FORTRAN program segment.

```
      INTEGER*2 LEN
      DATA MOD/2/
    1 CALL READ(REG,LEN,0,LNR,2,&2)
      CALL WRITE(REG,LEN,MOD,LNR,3)
      GO TO 1
    2 STOP
```

This program performs a SEQUENTIAL read and an INDEXED write using the line number from the read operation as the line number specification for the write operation. The command (assuming compilation of the above into -LOAD)

```
$RUN -LOAD 2=A 3=B
```

is equivalent to

```
$COPY A B@I
```

and copies file A into file B preserving the line numbers of file A as the line numbers for file B (assuming B is initially empty). If a series of I/O operations involving a given usage of a line file are intermixed with INDEXED and SEQUENTIAL operations, the SEQUENTIAL operation begins sequentially with the line following the last line specified in the INDEXED operation. INDEXED operations following SEQUENTIAL operations use the line number given in that INDEXED specification.

I/O operations involving sequential files must be done sequentially. If the user specifies INDEXED on a sequential file operation, an error message is generated unless the global switch SEQFCHK is OFF, in which case the operation is performed as if SEQUENTIAL was specified. Attempting a sequential operation with a starting line number other than 1 (for example, $COPY FYLE(2)) also gives an error comment if SEQFCHK is ON.

I/O operations involving sequential devices, such as card readers, printers, card punches, magnetic tape units, and terminals, are inherently sequential and are normally done sequentially. If the SEQUENTIAL modifier is specified, the line number attached to the line is the current value of the line pointer plus the increment specified on the FDname. If the INDEXED modifier is specified, the line number attached

to the line is the line number specified in the calling sequence. The INDEXED modifier is used primarily in conjunction with the PREFIX modifier. Note that the device treats the I/O operation as if SEQUENTIAL were specified.

Bit 29	EBCD	Value:	4 (dec)	00000004 (hex)
28	BINARY, BIN		8	00000008

Default: EBCD
Exceptions: None

The EBCD/BINARY modifier pair is device dependent as to the action specified. For card readers and punches, the EBCD modifier specifies EBCDIC translation of the card image which means that each card column represents one of the 256 8-bit EBCDIC character codes. The BINARY modifier specifies that the card images are in column binary format which means that each card column represents two 8-bit bytes of information. The top six and bottom six punch positions of each column correspond to the first and second bytes respectively with the high order two bits of each byte taken as zero. Printers and files ignore the presence of this modifier pair. The MEMOREX Transmission Control routines (that is, terminal DSR's) recognize this modifier pair. Note that card input through HASP (normal batch jobs) cannot be read in column binary.

For information on the usage of this modifier pair in specifications involving the devices listed above, see the respective User's Guides in Volume 4 of the MTS Manual. The list of device support routines recognizing this modifier is volatile and subject to change without notice. Users who wish to keep their programs device-independent should not specify this modifier.

Bit 27	LOWERCASE, LC	Value:	16 (dec)	00000010 (hex)
26	CASECONV, UC		32	00000020

Default: LOWERCASE
Exceptions: None

The LOWERCASE/CASECONV modifier pair is not device-dependent. If the LOWERCASE modifier is specified, the characters are transmitted unchanged. If the CASECONV modifier is specified, lower-case letters are changed to upper-case letters. This translation is performed in the user's core region. On input operations, the characters are read into the user's buffer area and then translated. On output operations, the characters are translated in the user's buffer area and then written out. Only the alphabetic characters (a-z) are affected by this modifier. Unlike IBM programming systems,

MTS considers the characters ¢, ", and ! as special characters rather than "alphabetic extenders" and thus the UC modifier does not convert ¢, ", and ! into @, #, and $, respectively.

Bit 25	NOCARCNTRL, NOCC	Value:	64 (dec)	00000040 (hex)
24	CC, STACKERSELECT, SS		128	00000080

Default: NOCARCNTRL
Exceptions: CC for PTR, TTY, 2741, 2260, SDA, and HPTR

The NOCC/CC modifier pair is device-dependent. This modifier pair controls the presence or absence of logical carriage control (or stacker-selection) on output records. For printer and terminal devices, the first character of each record is taken as logical carriage control if this character is a valid carriage control character and if the CC modifier is specified. If the first character is not valid as a carriage control character, the record is written as if NOCC were specified. For further information on logical carriage control, see the "Carriage Control" section in Volume 3. For card punches, the first character of each card image is taken as the stacker-select character if it is a valid logical stacker-select character (0, 1, or 2) and if the SS modifier is specified. If the first character is not valid as a stacker-select character, the card image is punched as if NOCC were specified. The SS modifier is intended only for those users who are communicating directly with a physical punch (normally system programmers) and is not implemented for normal batch usage under HASP. Note that the SS and CC modifiers reference the same modifier bit and thus may be used interchangeably.

The magnetic tape routines also recognize the presence of this modifier pair. For this description, see the "Magnetic Tape User's Guide" in Volume 4 of the MTS manual. Files ignore the presence of this modifier pair.

Bit 23		Value:	256 (dec)	00000100 (hex)
22	PREFIX, PFX		512	00000200

Default: ¬PREFIX
Exceptions: None

The PREFIX modifier pair controls the prefixing of the current input or output line with the current line number. On terminal input, the current input line number is printed before each input line is requested. The line number used is determined as specified in the description of the SEQUENTIAL and INDEXED modifiers. An example for terminal input is

```
$COPY *SOURCE*(6,,2)@PFX  A(6,,2)
      6_ first input line
      8_ second input line
          .
          .
end-of-file indicator
```

Note that this would have the same effect with respect to line numbering as

```
$GET A
$NUM 6,2
      6_ first input line
      8_ second input line
          .
          .
     xx_$UNN
```

The current (prefix) line number is not equivalent to the file line number. In the example above, the prefix line and the file line numbers were explicitly made to correspond by also specifying a line number range on the output FDname (the file A). On input from card readers and files, the PREFIX modifier has no effect. On terminal output, the current line number is printed before each output line is written. The line number used is determined as specified in the description of the SEQUENTIAL and INDEXED modifiers. An example for terminal output is

```
$COPY A(1,10)  *SINK*(100,,2)@PFX
    100_ first output line
    102_ second output line
          .
          .
```

Note again that the current line number is not equivalent to the file line number. On output to the printer or to a file, the PREFIX modifier has no effect.

If the INDEXED and PREFIX modifiers are given together for terminal output, the line numbers referenced by the INDEXED modifier are the same as those produced by the PREFIX modifier. As an example, consider the following FORTRAN program segment:

```
      INTEGER*2 LEN
      DATA MOD/Z00000202/    Turns on INDEXED and PREFIX
    1 CALL READ(REG,LEN,0,LNR,2,&2)
      CALL WRITE(REG,LEN,MOD,LNR,3)
      GO TO 1
    2 STOP
```

This program performs a read SEQUENTIAL and a write INDEXED and PREFIX. The command (assuming compilation of the above into -LOAD)

```
$RUN -LOAD   2=A 3=*SINK*
```

is equivalent to

```
$COPY A  *SINK*@I@PFX
```

which is also similar to

```
$LIST A
```

with a slightly different formatting of the line numbers.

Bit 21		Value:	1024 (dec)	00000400 (hex)
20	PEEL, GETLINE#, RETURNLINE#		2048	00000800

Default: ¬PEEL
Exceptions: None

The PEEL modifier pair has two functions depending upon whether it is specified on input or on output. On input, if the PEEL (GETLINE#) modifier is specified, a line number is extracted from the front of the current input line. The line number is converted to internal form (external value times 1000) and returned in the line number parameter during the read operation. See the subroutine description of SCARDS and READ. The remainder of the line is moved into the input region specified. As an example, consider the following FORTRAN program segment:

```
      INTEGER*2 LEN
      DATA MOD/2048/
    1 CALL SCARDS(REG,LEN,MOD,LNR,&2) Read with PEEL
      CALL SPRINT(REG,LEN,0,LNR)
      GO TO 1
    2 STOP
```

The program reads an input line, extracts the line number, and writes out the line without its line number. The following sequence (assuming compilation of the above into -LOAD)

```
$RUN -LOAD   SCARDS=*SOURCE* SPRINT=ABC
10AAA
12BBB
```

is equivalent to

```
$COPY *SOURCE*@GETLINE#  ABC
10AAA
12BBB
```

Listing the file ABC produces

```
$LIST ABC
     1     AAA
     2     BBB
```

If the PEEL modifier is specified on input in conjunction with the INDEXED modifier on output, the line number of the input line can be used to control the destination of the line during output. For example:

```
      INTEGER*2 LEN
      DATA MOD1/2048/, MOD2/2/
    1 CALL SCARDS(REG,LEN,MOD1,LNR,&2) Read with PEEL
      CALL SPRINT(REG,LEN,MOD2,LNR)    Write INDEXED
      GO TO 1
    2 STOP
```

This program reads an input line, extracts the line number, and writes out the line with the extracted line number as the line number specification for an indexed write operation. The following sequence (assuming compilation of the above into -LOAD)

```
$RUN -LOAD  SCARDS=*SOURCE* SPRINT=ABC
10AAA
12BBB
```

is equivalent to

```
$COPY *SOURCE*@GETLINE#  ABC@I
10AAA
12BBB
```

which is also equivalent to

```
$GET ABC
10AAA
12BBB
```

Listing the file ABC produces

```
$LIST ABC
    10     AAA
    12     BBB
```

On output, if the PEEL (RETURNLINE#) modifier is specified, the line number of the current output line is returned in the line number parameter of the subroutine call during the write

operation. See the subroutine descriptions of SPRINT, SPUNCH, SERCOM, and WRITE. The line itself is written out and is unaffected by the presence or absence of this modifier. The modifier is used on output to aid the programmer in recording the line number of the current line written out.

Bit 19		Value:	4096 (dec)	00001000 (hex)
18	MACHCARCNTRL, MCC		8192	00002000

Default: ¬MCC
Exceptions: None

The machine carriage control modifier pair is device-dependent. The MCC modifier is used for printing output (via printers or terminals) from programs producing output in which the first byte of each line is to be used as the command code in the Channel Command Word (CCW) used for output to a 1403 printer. If the MCC modifier is specified and the first byte of the output line is a valid 1403 CCW command code, the line is spaced accordingly and printing starts with the next byte as column 1. If the first byte is not a valid 1403 CCW command code, the entire line is printed using single-spacing. Spacing operations performed by machine carriage control occur <u>after</u> the line is printed (as opposed to logical carriage control in which the spacing is performed <u>before</u> each line is printed). Most programs do not produce output using machine carriage control. The few programs that do (such as *ASMG and the TEXT360 programs) internally specify MCC for their output assuming that it is bound for a printer. Hence MCC need not be specified. If the user directs the output to a file, MCC must be specified when the file is printed. For example:

```
$RUN *ASMG SCARDS=A SPRINT=B SPUNCH=C
$COPY B TO *SINK*@MCC
```

The MCC modifier pair is ignored for files and all devices other than printers or terminals connected through the MEMOREX Transmission Controller. For further information on machine carriage control, see the "Carriage Control" section in Volume 3.

Bit 17		Value:	16384 (dec)	00004000 (hex)
16	TRIM		32768	00008000

Default: ¬TRIM
Exceptions: TRIM for line files, sequential files, and HPTR

The TRIM modifier pair is used to control the trimming of trailing blanks from input or output lines. If the TRIM

modifier is specified, all trailing blanks except one are trimmed from the line. If ¬TRIM is specified, the line is not changed. A trimming operation does not physically delete the trailing blanks from the line, but only changes the line length count.

Bit 15		Value:	65536 (dec)	00010000 (hex)
14	SPECIAL, SP		131072	00020000

Default: ¬SP
Exceptions: None

The SPECIAL modifier pair is reserved for device-dependent uses. Its meaning depends upon the particular device type with which it is used. The only device support routines recognizing this modifier pair are the sequential file routines.

For information on the usage of this modifier pair see Appendix B of this volume. The list of device support routines recognizing this modifier is volatile and subject to change without notice. Users who wish to keep their programs device-independent should not specify this modifier.

Bit 13		Value:	262144 (dec)	00040000 (hex)
12	IC		524288	00080000

Default: The setting of the IC global switch (usually ON)
Exceptions: None

The IC modifier pair controls implicit concatenation. If the IC modifer is specified, implicit concatenation occurs via the "$CONTINUE WITH" line. If ¬IC is specified, implicit concatenation does not occur. For example, $LIST PROGRAM@¬IC lists the file PROGRAM and prints "$CONTINUE WITH" lines instead of interpreting them as implicit concatenation. The use of the IC modifier in I/O subroutine calls or as applied to FDnames overrides the setting of the implicit concatenation global switch (SET IC=ON or SET IC=OFF) for the I/O operations for which it is specified.

Bit 1	ERRRTN	Value:	1073741824 (dec)	40000000 (hex)

Default: ¬ERRRTN
Exceptions: None

If the ERRRTN modifier is specified (bit 1 in the modifier word is 1) when an I/O call is made, and if an I/O error occurs when no SETIOERR/SIOERR interception has been established, the error return code is passed back to the calling

program instead of causing an error comment to be printed. This modifier may be used only with an I/O subroutine call. It may not be used as an attribute on an FDname.

Bit 0 NOTIFY Value: -2147483648 (dec) 80000000 (hex)

Default: ¬NOTIFY
Exceptions: None

If the NOTIFY modifier is specified (bit 0 in the modifier word is 1) when an I/O subroutine call is made, on return GR0 is set to a value indicating what has happened:

0 = no unusual occurrence
1 = new FDUB opened and no I/O done
2 and above, reserved for future expansion

A new FDUB is opened if implicit concatenation occurs if a change to the next member of an explicit concatenation is effected, or if a replacement FDname is requested. This modifier may be used only with an I/O subroutine call. It may not be used as an attribute on an FDname.

April 1974

APPENDIX B: SEQUENTIAL FILES AND NOTE AND POINT

Associated with every sequential file are at least three logical pointers which determine where the next read or write operation starts. Every sequential file has one Read Pointer, one Write Pointer, and one Last Pointer for each usage of the file.[1] These logical pointers are automatically updated after every read or write operation by the file routines as outlined below. In addition, two subroutines, NOTE and POINT, callable from assembly language and FORTRAN, "remember" the current values of these logical pointers, and, at some later time, "alter" the values of these pointers. In so doing, a user is able to start reading and/or writing a sequential file from points other than the beginning and/or end of the file.

These three logical pointers are manipulated by the file routines as follows:

The Read Pointer is always initially set to point to the beginning of the file when the file is created or first referenced. The Read Pointer is updated after every read operation to point to the next line to be read. A particular Read Pointer affected by a rewind operation (that is, the Read Pointer associated with the FDUB on which the rewind was given) is reset to point to the beginning of the file. Finally, all Read Pointers are reset to point to the beginning of the file whenever the file is emptied.

The Write Pointer is initially set to point to the beginning of the file when the file is created, and is updated after every write operation to point to the next line to be written. The Write Pointer is reset to point to the beginning of the file when the file is emptied or rewound. In addition, if after any read operation, the Read Pointer is greater than the Write Pointer, the Write Pointer is updated to coincide with the Read Pointer. This allows a user to rewind a sequential file, begin reading, stop at some intermediate point and begin writing at that point. This is similar to what would happen if the same operations were performed on a magnetic tape. The difference is that, if after writing a few lines, the user again began to read the file, he would begin reading from the intermediate point at which he previously stopped reading and started writing (that is, the Read Pointer is not updated after a write operation). Finally, whenever the file is opened, the Write Pointer is set equal to the Last Pointer.

The Last Pointer is initially set to point to the beginning of the file when the file is created, and is updated after every write operation to

---

[1]When a user has more than one logical I/O unit attached to the same file, or has called the subroutine GETFD more than once for the same file, MTS creates a FDUB (File or Device Usage Block) for each of these usages, and each FDUB points to the one File Control Block (FCB) for the file. (This is not the case when two separate users each have a single usage attached to the same (shared) file. In this case each user has a FDUB pointing to his own FCB for the given file.) The ramifications of more than one Read Pointer are discussed later.

coincide with the new updated Write Pointer. The Last Pointer is also considered the logical end of file, so that writing a file beginning from some intermediate point implies that any information from that point on is to be discarded. However, if the write operation had the @SP modifier specified, the Last Pointer is not set to the Write Pointer after the operation, and the rest of the information is still there. In this case, the record being written into the file must have the same length as the one that was there before. If it is not the same length, it is truncated or padded with blanks as necessary, it is written into the file, and then an error message (interceptable as usual with the @ERRRTN modifier or the SETIOERR subroutine) is issued. Finally, the Last Pointer is reset to point to the beginning of the file whenever the file is emptied.

The NOTE subroutine can be called to obtain the values of these pointers when reading and/or writing a sequential file, and then the POINT subroutine can be called to "reposition" these pointers. (See the subroutine descriptions in Volume 3 for calling sequences.) A call to NOTE obtains four fullwords of information to be used later. These four fullwords are respectively, the Read, Write, and Last Pointers, as well as the last line number (useful only for sequential-with-line-number files)[2] associated with the file corresponding to the FDUB-pointer given. These pointers always correspond to the next line about to be read or written. At some later point in time, the caller is able to indicate which of these pointers he wishes to alter for the next read or write operation and, using the information returned by NOTE, can call the POINT subroutine appropriately.

The logical pointers are valid from one copy of a sequential file to another as long as both copies are located on the same kind of storage. Thus, the pointers returned by NOTE for a file on the disk can be used to point into an identical copy of the file elsewhere on disk storage.

The user should be aware of the consequences of reading and/or writing multiple logical I/O units or other usages attached to the same file. Since there is one Read Pointer associated with each FDUB-pointer and with each logical I/O unit, the user is able to read alternately from a number of different points in the file, overlapping or not as the application dictates. However, since there is only one Write Pointer and one Last Pointer associated with the file, writing to multiple usages attached to the same file amounts to simply appending to the end of the file in the order that the write requests are received.

---

[2]The rationale for wanting to remember the last line number in a sequential file with line numbers is that when writing this type of file, the caller must insure that line numbers are in numerically increasing order.

April 1974

APPENDIX C: INTERNAL FILE STRUCTURE AND THE SIZE OF FILES

In order to attempt some explanation of how big a file must be to hold a specified amount of data, a certain amount of detail about the internal structure of a file is needed.

The basic unit of a file is a physical record. Files (both line and sequential) on the 2314 disks are written two records per physical track; each record is 3520 bytes long. Since disk space is allocated to files in tracks, each disk file contains an even number of records.

At this point, the explanation must be split into two categories -- line files and sequential files, since the allocations of records in these two types of organization differs considerably.

Line Files

A line file contains three logical components: the track index, the line directory, and the data section.

The track index is used to associate a physical disk address (disk pack volume name, cylinder, track, and record address) with a logical record number (integer between 1 and 218). In the line directory, all references are to a logical record number. This saves space and makes the addressing relative.

The line directory is fixed-length entries, one for each line in the file, and ordered by line number. There are also entries for each available "hole" in the data section. These entries contain the line number, the length of the line, and indicate where that line is, in terms of logical record number and offset within the record. The line directory only points to the lines themselves, which are in the data section.

The data section contains the actual lines, unordered and unpacked (that is, holes that result because of line replacements stay there until a piece that length or less is needed).

If the file is small enough, the track index and line directory may be contained in one physical record. When the file gets larger than this limit, then it is called "extended", and the track index occupies one record and the line directory occupies one or more records.

Now to get some approximate numbers for sizes of line files.

If n is the number of lines in the file,
r is the average length of a line,

then the number of records needed:

R = data section + line directory + track index

which is approximately

$$R = \left\lceil \frac{n}{[3520/r]} \right\rceil + x + 1$$

where x = 0 if the file is not extended (line directory is with track index)

and x = [n/439] if extended (439 8-byte entries in a 3520 byte record less 8-byte header) and if not an integer, the next larger integer.

The criterion for extension is

track index bytes + line directory bytes > 3520 - 8-byte header

or 16*R + 8*n > 3512

or n > 439-2*R

so that if n > 435, the file is definitely extended.

The number of tracks needed is T = [R/2]

The number of pages charged is P = T*7294/4096

The above figures are approximate, and closest only when the file is first filled. In particular, the parameter n above really is the number of lines plus the number of holes, and, after the file has undergone much editing, it may be considerably greater than the number of lines. Copying a file into another file, emptying the original file, and then copying it back condenses the file. This procedure is recommended after a file has been extensively edited.

Sequential Files

The organization of sequential files (with or without line numbers) is quite simple compared to line files. In general, the first n bytes (currently n=215) of the first record are used as a header in which pertinent information about the sequential file is retained.

Immediately following this header information are the lines of information stored in the sequence in which they were received. Since these lines may be up to 32767 bytes long, and since the physical records on the disk are 3520 bytes, it is quite possible that a line has to be broken up and stored on more than one physical record. Since the lines are packed end to end using up all of one physical record before going on to the next physical

record, this is quite likely, even if only "short" lines are written into a sequential file. Thus it turns out that even short lines may be broken across physical record boundaries.

For this reason it is convenient to refer to a segment of a line as that part of the line which resides on a physical record. A line can therefore consist of one segment, two segments, or more, depending on the size of the line, the size of the physical records, and how the line "fell" with respect to physical record boundaries. Each segment of a line in a sequential file has either 10 or 6 bytes of overhead associated with it depending on whether it is in a sequential file with or without line numbers.

This gives the following lower bound for the number of records:

$$R = \left\lceil \frac{215 + n*(r+k)}{3520} \right\rceil$$

where k=6 if SEQ
=10 if SEQWL

The number of tracks needed: $T = [R/2]$

The number of pages charged: $P = T*7294/4096$

This is a lower bound because it does not consider that when a line is spanned across record boundaries, each of the segments has the k bytes of overhead information attached to it, and on the average, probably every record boundary has a line spanned across it.

All of the numbers in this appendix are subject to change without notice.

APPENDIX D: I/O ROUTINES RETURN CODES

The return codes that may result from a call on an input or output subroutine depend on the type of the file or the device used in the operation. In general, a return code of zero means successful completion of the input or output operation, and a return code of 4 means end-of-file for an input operation and end-of-file-or-device for an output operation. If the file or device being used was specified as part of an explicit concatenation (and is not the last member of that concatenation), a return code of 4 causes progression to the next element of the concatenation, and that return code is not passed back to the caller. For example, if

```
SCARDS=A+B
```

then when the call is made to the SCARDS subroutine after the last line in A has been read, the file routines signal an end-of-file, but this is intercepted, and the first line in B is read.

Return codes greater than 4 are normally not passed back to the caller but instead cause an error comment to be printed and control to be returned to command mode. There are two ways to suppress this action and gain control in this situation. First, the subroutines SETIOERR and SIOERR (see descriptions in Volume 3) are provided to permit a global intercept of all input/output errors. Secondly, specifying the ERRRTN modifier on an I/O subroutine call causes all return codes to be passed back.

A description of the return codes that may occur with a particular file or device is found in the appropriate User's Guide. In addition, a summary is provided below. Non-zero return codes marked with an asterisk are fatal error returns; the others are considered errors by MTS, but are not necessarily fatal.

Files:

Input	0	Successful return
	4	End-of-file (sequential read) Line not in file (indexed read)
	8*	Error
Output	0	Successful return
	4*	Size of file exceeded
	8*	Line numbers not in sequence (SEQWL)
	12*	For future expansion, should not occur
	16*	For future expansion, should not occur
	20*	Sequential file written with indexed modifier, or written with starting line number other than 1
	24*	System error
	28*	Hardware or system error
	32	Line truncated (@SP on sequential file)
	36	Line padded (@SP on sequential file)

Magnetic tape:

Input	0	Successful return
	4	Tape-mark (end-of-file) sensed on read, BSR, or FSR operation
	8	Load point reached on BSR or BSF control command
	12	Logical end of labeled tape reached on read, FSR, or FSF operation
	16*	Non-recoverable hardware I/O error or invalid control command parameter
	20*	Should not occur
	24*	Fatal error (may be due to hardware malfunction, label error in which the position of the tape is uncertain, or pulling the tape off the end of the reel during a read, FSR, or FSF operation); following a fatal error, the tape must be rewound before any other I/O operation is allowed
	28*	Volume or data set in error
	32*	Sequence error caused by issuing a control command when the tape is not positioned properly; or a read, FSR, or FSF operation following a write operation
	36*	Deblocking error caused by improper blocking parameters, for example, attempting to deblock a format FB file using a format VB specification
Output	0	Successful return
	4	End-of-tape marker sensed during write or WTM operation
	8	Load point reached on BSR or BSF control command
	12*	Attempt to write more than 5 additional records after end-of-tape marker sensed
	16*	Non-recoverable hardware I/O error, invalid control command, or invalid control command parameter
	20*	Attempt to write on file-protected tape
	24*	Fatal error (may be due to hardware malfunction, label error in which the position of the tape is uncertain, or pulling the tape off the end of the reel during a read, FSR, or FSF operation); following a fatal error, the tape must be rewound before any other I/O operation is allowed
	28*	Volume or data set in error
	32*	Sequence error caused by issuing a control command when the tape is not positioned properly; or a read, FSR, or FSF operation following a write operation
	36*	blocking error caused by improper blocking parameters which are inconsistent with the labels of the file being written

Card input under HASP:

Input	0	Successful return
	4	End-of-file
	8*	Attempt to read in column binary mode

Output	8*	Attempt to write on card reader
Printed Output:		
Input	8*	Attempt to read from printer
Output	0	Successful return
	8*	Local page limit exceeded (User never gets control back for global limit exceeded.)
Punched Output:		
Input	8*	Attempt to read from punch
Output	0	Successful return
	8*	Local card limit exceeded (User never gets control back for global limit exceeded.)
MERIT Network:		
Input:	0	Successful return
	4	End-of-file read from network. This does not necessarily mean that there is no more data to be read from the network, only that the remote host has sent an end-of-file.
	8*	Read not allowed; must write. This means that the remote host is requesting input from the network and, to avoid a deadlock, the local program must not read from the network. The prompting characters sent by the remote host when it did the read are returned to the user.
	12*	Should not occur
	16*	Connection is closed; no I/O may be done.
Output	0	Successful return
	4*	Should not occur
	8*	Write not allowed; must read. This means that the remote host has issued a write on the network and, to avoid a deadlock, the local program must not write on the network.
	12*	Should not occur
	16*	Connection is closed; no I/O may be done.
Control	0	Successful return
	4*	Error in control command for local host
	8*	Control command not allowed -- the remote host has not done a read.
	12*	Should not occur
	16*	Incorrect control command syntax for remote host
	20*	Invalid syntax or context for control command
Most other devices:		
Input	0	Successful return
	4	End-of-file
	8*	Error

Output	0	Successful return
	4	End-of-file-or-device [if applicable]
	8*	Error

APPENDIX E: UPDATING FILES DEFENSIVELY IN MTS

One of the hazards of using a time-sharing system like MTS is the unfortunate fact that when machine problems or power failures cause the system to go down abruptly, terminal users are left holding the phone with no recourse for recovery of work currently in progress. This is particularly frustrating if the terminal user happens to be in the process of updating a file when the system goes down. Due to the structure of files in MTS, it is quite possible that a random portion of the most recent changes he initiates may not be made. In addition, if the terminal user is particularly unfortunate, his partially updated file may possibly be left in an inconsistent, unusable state by the abrupt halting of his terminal session.

Two questions often asked are: Why are some changes made before a system crash and not in the users file after the system comes up? What can the terminal user (or even an interactive program) do to minimize the man-hours lost when the updating process is terminated prematurely due to machine or power failures?

To understand the answer to the first question (why changes are not really made), some background about the file system in MTS is necessary.

When a file is first referenced, the first few physical blocks (physical units of information, in general containing many lines) of the file are read from disk and placed into buffers (blocks of storage in main memory). This is done because transferring information to and from the user via these buffer images of the physical blocks is much more efficient than actually reading and writing physical blocks of secondary storage at every request. This process of reading the first few physical blocks into buffers when the file is first referenced has historically been called "opening the file." When MTS is requested to read a line (logical record) from the file and pass it to the user, if that line happens to be in a buffer, the information is immediately passed to the requester. If the line is not in the buffer, an existing buffer must be "emptied" and the physical block on secondary storage containing the requested line must then be read from disk into the now empty main memory buffer and then, as before, the line requested is passed to the user.

Similarly, when MTS is requested to take a line from the user and write it into a file, if that line "happens to fit" (both physically and logically) into a buffer in main memory, it is put directly into the appropriate place in the buffer. If it doesn't happen to fit into any buffers, an existing buffer must be "emptied" and the physical block from secondary storage into which the line properly goes is read into the empty buffer and the line "fitted" into the appropriate place. In either case, once the line is "written", the buffer into which it was placed is flagged as changed. This means that the buffer in main memory is no longer identical to the corresponding physical block on secondary storage and, more importantly, the buffer image in main storage is henceforth the one and only valid representation of that block of the file.

The problem is that while a file is open, its true and valid state is reflected not only in its physical blocks on secondary storage but also by its "quite possibly changed" buffer images in main memory. When the system goes down abruptly, all buffer images in main memory are lost.

Thus the big question appears to be: When are buffer images in main memory written out in updated form on secondary storage? The answer is that while the file is open the buffers are written out to secondary storage only when necessary. Whenever a new physical block is needed in main memory and a buffer is not available, an existing buffer must be emptied. Before that buffer is emptied and reused, however, a check is made to see if the buffer has been changed, and, if it has, the buffer is written back on secondary storage in updated form. There is no direct relationship between the time changes are made to a buffer and the time the buffer is written in updated form on secondary storage.

When the user has finally finished updating the file, MTS writes all main memory buffers that have been changed in updated form onto secondary storage. Only at this point is the file safely on secondary storage in a consistent and up-to-date state. This process of writing changed buffers onto secondary storage historically has been called "closing the file".

Turning to the second question, it becomes evident that the one thing a terminal user (or interactive program) can do to minimize loses when the system goes down during the updating process is to request MTS to close a file more often than it otherwise might. This brings up the question, when does MTS normally close a file? To minimize loses, MTS closes all open files before the execution of every command. To answer the second question, if a terminal user is adding information to a file over long periods of time using automatic numbering, it would be expedient to occasionally do a $COMMENT command. This would at least guarantee that lines entered up to the point of the $COMMENT command were properly written on secondary storage.

If a file is being updated from a terminal using the context editor ($EDIT), the editor command "MTS $EDIT" returns to MTS command mode and then returns to the editor. This ensures that the file being edited is properly written to the point of the $EDIT command. (Note that the commands "CHECKPOINT" and "RESTORE" are of no help in the event of system crashes, since they are meant to be used only if the user changes his mind and wishes to backtrack.)

The MTS subroutine FREEFD (the counterpart of GETFD) also closes the indicated file. Thus, interactive user programs which write large amounts of information into a file over long periods of time should occasionally call FREEFD (and GETFD again) to ensure that changes made up to that point in time are actually accomplished. In particular, programs which complete the updating process but continue to execute for long periods of time should always call FREEFD as soon as updating is complete, since MTS does not close the file until execution has terminated.

These procedures allow a user or program to force MTS to close a file more often than it otherwise might, and in that way do much to minimize the

man-hours of work lost if the system goes down before the updating process is completed. There is not much that can be done to salvage the changes that were made between the time the file was last closed and the time the system went down. Generally the only lose is that of checking and redoing those changes (not necessarily the most recent) entered but not accomplished during the last open period. If one is unlucky, his file (open when the system crashed) will show signs of inconsistency which are not correctable through normal means. These inconsistencies are evidenced by, for example, two lines with the same line number, or lines which refuse to be deleted or altered. Some inconsistencies, however, make a file totally unreadable. In cases of this nature, Academic Services should be consulted, and if the damage is beyond human repair, the file will have to be restored from a save-tape to its status of the day before. In cases where many and time-consuming updates are being made over long periods of the day and the restoring of a file to its status of the day before cannot be tolerated, the user should probably make changes to a copy of the file and periodically update the original by $COPYing the copy.

Some general comments can be made in closing. First, upon reflection, it should be evident that if a file is only being read, no problems can result from a system crash since none of the main memory buffers are changed and thus the file on secondary storage is always valid. Second, due to their internal structure, line files are much more susceptible than sequential files to inconsistencies caused by system crashes during an updating process. Thus one should use sequential files if at all possible. Finally, update defensively. The work you save is your own.

April 1974

SYSTEM COMMAND LANGUAGE

The sequence of operations of a job run in MTS is controlled by the commands of the system command language. A command is a request for the system to perform a function such as running a program, creating or destroying a file, or setting a global switch.

The system command language provides several modes of operation. The basic mode is MTS command mode. Several other modes also exist which are logically separate modes within the system command language. This means that each mode may be entered or left at any time without affecting the status of any other mode. The other modes currently available are:

(1) execution mode which is running a user program,
(2) edit mode which is supported by the context editor,
(3) debug mode which is supported by the symbolic debugging system, and
(4) network mode which interfaces the MERIT computer network.

The interpretation of a line entered by the user at a terminal or of a card read from a batch job depends on which mode is in control and reading the input line. If the MTS monitor is in control, the system is in MTS command mode. This is the normal mode of operation for the system. In MTS command mode, the system is waiting for the next MTS command to process or the next line to enter into the currently-active file. If the context editor is in control, the system is in edit mode; if the symbolic debugging system is in control, the system is in debug mode. If the user is running a program, then his program is in control and he is in execution mode. The interaction of these modes is discussed below.

PREFIX CHARACTERS

In order that the user can determine the mode of the system, a prefix character is printed at the front of all input or output lines at the terminal to indicate "who is speaking" or "who is listening". On output lines, this prefix character is typed ahead of the message. When input is requested, either the prefix character (if automatic line numbering is off) or the prefix character followed by the line number (in MTS command mode if automatic line numbering due to the NUMBER command is on) is typed at the front of the line. Each mode of the system has a distinct prefix character. For example, MTS command mode uses the pound sign "#". Thus, whenever the MTS monitor is expecting a command line or data line to place in the currently-active file, it types a "#" in the first character position of the terminal line and waits for input. The system prefix characters are:

Mode	Prefix Character
MTS command mode	#
Copying	>
Loading	.
Prompting	?
Edit command mode	:
Fast insertion	?
Debug command mode	+
At-insertion	%
Network command mode	)
Execution mode	blank

In addition, many of the programs available in public files use prefix characters. For example, PIL uses "=". The execution mode prefix character can be changed by the subroutine SETPFX which is described in Volume 3. The printing of prefix characters can be suppressed by the command

```
$SET PFX=OFF
```

INTERACTION OF MODES

The following is a description of the interaction of the various modes. The normal mode is MTS command mode (sometimes called just "command mode"). The user always starts in this mode, and reverts to it in case of trouble.

MTS Command and Execution Mode

The user may enter execution mode via the RUN, START, or RESTART commands. When the user is in execution mode, the executing program is in control. The return to MTS command mode is normally accomplished by:

(1) the program returning to MTS (or calling the subroutines SYSTEM, MTS, MTSCMD, or ERROR);
(2) the program exceeding a local time, page, or card limit;
(3) in batch use, the program exceeding the global time, page, or card limit;
(4) in conversational use, the user issuing an attention interrupt, unless the program being executed has called the appropriate subroutines to intercept attention interrupts; or

(5) some abnormal condition (such as a program interrupt) occurring and the program has not called the appropriate subroutines to intercept the condition.

MTS Command and Debug Command Mode

The user may enter debug command mode via the DEBUG or SDS commands. When the user is in debug mode, the symbolic debugging system is in control. If a "one-shot" debug command is specified with the SDS command, that debug command is executed in debug mode and then an immediate return is made to MTS command mode. The user may return to MTS command mode via the MTS or STOP debug commands. If STOP is used, the storage used by the symbolic debugging system is released.

Debug Command and Execution Mode

The user may enter execution mode from SDS via the RUN, CONTINUE, GOTO, or STEP debug commands. A return is made from execution mode to debug command mode in the same manner as from execution mode to MTS command mode (see above). Also, a return is made to debug mode when any of the following conditions occur:

(1) encountering a breakpoint or at-point in the program;
(2) completion of the step count for the STEP command;
(3) calling the subroutines LINK, LOAD, or XCTL when the XFR debug option is ON.

If the SET DEBUG=ON option has been specified, the user may enter execution mode indirectly through debug mode from MTS command mode via the RUN, START, or RESTART commands. The return from execution mode is made through debug mode to MTS command mode.

MTS Command and Edit Command Mode

The user may enter edit command mode via the EDIT command. When the user is in edit command mode, the context editor is in control. If a "one-shot" edit command is specified with the EDIT command, that edit command is executed in edit mode and then an immediate return is made to MTS command mode. The user may return to MTS command mode via the MTS or STOP edit commands. If STOP is used, the file being edited is released by the editor.

MTS Command and Network Command Mode

The user may enter into network command mode via the NET command. When the user is in network command mode, the user is interfaced with the MERIT computer network. The user may return to MTS command mode via the MTS or STOP network commands. If STOP is used, the storage used by the network interface is released.

Copying

Copying is an extension of MTS command mode. Copying is in effect when a COPY or LIST command has been given in MTS command mode. When copying, the user is reading lines from one file or device and writing them on another file or device. When an end-of-file condition is received from the file or device being read from, copying stops. In conversational use, copying also stops if the user issues an attention interrupt.

In order to copy a $ENDFILE line from *SOURCE* or *MSOURCE*, a $SET ENDFILE=NEVER command must be issued first. In this case, the terminal user must use an end-of-file terminal control character to terminate the copying. The batch user must normally make this COPY or LIST command the last command in the card deck, since in this situation, the copying proceeds until the end of his deck or until the line $CONTINUE WITH *DUMMY* is encountered.

Loading

Loading is an extension of MTS command mode. When a RUN, LOAD, or DEBUG command is given, the dynamic loader takes control to load the specified program. This is a temporary condition. After the program is loaded, the user is then in execution mode (for a RUN command), MTS command mode (for a LOAD command), or debug mode (for a DEBUG command). All error messages from the loader and the load map, if requested, are printed by the dynamic loader.

Prompting

This is an extension of MTS command mode for conversational use. When the prefix character "?" appears at the front of an input line, the user must enter the information that was requested on the previous output line. This usually occurs because of an erroneous or incomplete command or when confirmation is requested for a DESTROY or EMPTY command.

COMMANDS AND DELIMITERS

Commands are interpreted as such by MTS only if MTS is in MTS command mode. Thus a line

```
$COPY  A B
```

is treated as a copy command if the MTS monitor reads the line, but is treated as a data line if the user's program reads it during execution mode.

Delimiters are constructs which are recognized in a broader context than commands. Because these delimiters start with a dollar sign "$", they are often erroneously considered to be commands. Since they are not commands, the dollar sign is <u>not</u> optional, and there are no abbreviations for them.

There are currently two delimiters: the end-of-file delimiter and the implicit concatenation delimiter.

The form of the endfile delimiter is

$ENDFILE

Sensing this delimiter causes an end-of-file condition to be signalled. The scope of this delimiter depends on the setting of the global ENDFILE switch which can be set by the MTS SET command or by the CUINFO subroutine. (See the GUINFO, CUINFO subroutine description in Volume 3.) In the default case, ENDFILE=OFF, a line consisting of "$ENDFILE" is recognized as a delimiter only if it is read from *SOURCE* or *MSOURCE*. If the SET command specifies ENDFILE=ON, a line consisting of "$ENDFILE" is always treated as an end-of-file delimiter. If ENDFILE=NEVER is specified, a line consisting of "$ENDFILE" is never treated as a delimiter, but is always treated as a data line.

The form of the implicit concatenation delimiter is

$CONTINUE␣WITH FDname [RETURN]

where ␣ means exactly one blank. A line with this construct, if recognized as a delimiter, causes implicit concatenation to occur. For a description of implicit concatenation, see the "Files and Devices" description in this volume. Recognition of a delimiter is controlled by both a global switch and a pair of I/O modifier bits. The global switch is IC and its setting is specified by the SET command. Its values are either ON or OFF. The default case is IC=ON which means that this construct is recognized as a delimiter. The I/O modifiers are IC and ¬IC (see "Files and Devices") and when applied to an FDname, they override the setting of the global IC switch for those usages of the FDname.

April 1974

MTS COMMAND MODE

MTS command mode is the default mode for the system. In MTS command mode, the MTS monitor is looking for the next command to process or the next data line to enter into the currently active file. It is in MTS command mode that the user initially begins his job and issues the various commands for running his programs. It is to MTS command mode that the user returns after his programs have terminated execution (either normally or abnormally).

In MTS command mode, input lines are read from *SOURCE*. *SOURCE* refers initially to the user's terminal or to the card reader for batch jobs submitted. This initial assignment may be changed by using the $SOURCE command.

COMMAND LINES AND DATA LINES

All lines read in MTS command mode are either

(1) interpreted as commands, or
(2) written into the currently active file.

The currently active file is established with either a $GET command (if it already exists) or a $CREATE command (if it does not exist). Another $GET or $CREATE establishes a different active file; the $RELEASE or $DESTROY command removes it.

A single dollar sign "$", occurring as the first character in the input line from *SOURCE*, is used to indicate that the line is a command line. If an input line does not begin with a single "$", it may be interpreted by MTS as a data line. Data lines are put into the currently active file. Every data line must have a line number associated with it. This is accomplished in MTS in two ways: either MTS supplies the line number (automatic line numbering on), or the user supplies the line number himself (automatic line numbering off).

In the first method (automatic line numbering on), MTS automatically assigns a line number to each input line. Automatic numbering is enabled by the $NUMBER command. Any line which is entered is interpreted as a data line unless the first character is a single "$" in which case MTS interprets the line as a command. For example:

```
$COMMENT
```

is interpreted as a $COMMENT command and the line is not placed into the active file. On the other hand the line:

```
COMMENT
```

is treated as a data line. It is given the line number which appeared as the terminal prefix number and placed in the active file. To actually place the line "$COMMENT" into the active file, the line:

```
$$COMMENT
```

must be specified. The first dollar sign is removed from the line before it is given a line number and placed into the active file.

In the second method (automatic line numbering off), the user must supply a line number with each line entered. This line number is specified by placing a legal MTS line number starting at the first character position on the line. A legal MTS line number is a decimal or integer number between -99999.999 and +99999.999 inclusive. MTS scans the line for a line number and if present, strips it from the line, and places the characters following the line number into the active file at the indicated line position. For example:

```
15HI THERE
```

places the characters "HI THERE" into the active file as line 15. The line number separator character "," may optionally be used to separate the the line number from the rest of the line when the line number and the line might be confused. For example:

```
21.6,10312 CONTINUE
```

would place the line "10312 CONTINUE" at line position 21.6 of the active file. If the user wishes to enter a line into the active file beginning with a ",", he must either specify two commas or change the line number separator character with the SET command. For example, either

```
25,,ABCDE
```

or

```
$SET LNS=|
25|,ABCDE
```

enters the line ",ABCDE" into line position 25 of the active file.

Users are cautioned that a line is still treated as a command if the first character after the line number (ignoring a line number separator) is a single "$" and the second character is something else. For example:

```
10,$COMMENT
```

is treated as a COMMENT command and:

```
10,$$COMMENT
```

must be specified if the line "$COMMENT" is to be placed at line 10 of the active file.

The following table summarizes the effects of entering a line with zero, one or two dollar signs.

	No $	One $	Two $
Line numbering on (terminal or batch)	Data line	Command line	Data line
Line numbering off with no active file (terminal)	Command line	Command line	Data line
Line numbering off with an active file (terminal)	Command line	Command line	Data line
Line numbering off (batch)	Data line	Command line	Data line

A line that starts with a single "$" is always treated as a command. A line that starts with two dollar signs "$$" is always treated as a data line. If no "$" is present the line is treated as a command for a terminal user if and only if

(1) there is no line number at the beginning of the line and automatic line numbering is off, or
(2) there is no active file.

In these cases the dollar sign is still optional but is completely ignored. If there is an active file and automatic numbering is on, then a dollar sign is required to distinguish command lines from data lines for the active file; if a line number is typed at the front of the line, a dollar sign is required for the same reason. The dollar sign is required in batch mode to avoid interpreting data as commands and causing irreversible damage to files.

If there is no currently active file and the line cannot be interpreted as a command, it is ignored and the comment "INVALID COMMAND" is given.

CONTINUATION LINES

If the last character of an input line from *SOURCE* being read by the MTS monitor is a minus sign "-", then the next input line is assumed to be a continuation of the current line. 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. In batch, the last character position is column 80 on the card. For a terminal input line, the "-" must be the last character typed before the termination of the line.

GLOBAL AND LOCAL LIMITS

In order to prevent run-away jobs in batch mode, it is necessary to provide some means for the user to limit the CPU time a job may use and the amount of output (paper and cards) it produces. It is often desirable for users at a terminal to be able to limit the execution time and output of a single program. To provide this facility, global and local limits are provided.

Global limits apply for the entire job from sign-on to sign-off. They are valid for batch jobs only and may be specified on the SIGNON command. Default values are assumed for any omitted limit specifications. There are no global limits imposed for terminal jobs.

Local limits apply for the execution of a single program. They may be specified on the $RUN, $LOAD, $RESTART, or $START commands. If a $RESTART or $START command specifies no limits, then what is remaining of the limits specified by the original $RUN or $LOAD command is used. If no limits are imposed for the execution or output of a program, then the program is not limited in terminal mode and limited only by the global limits in batch mode.

The first limit to be exceeded, whether global or local, causes an error comment to be printed and the job terminated (if a global limit), or a return to command mode (if a local limit).

A program storage dump is given if a limit has been exceeded in batch mode while executing a program and if the dump has been requested by the $ERRORDUMP command, the ERRORDUMP keyword parameter on the $SET command, or a call to the CUINFO subroutine. The CPU time and pages used in producing the dump are not included in the global limit specifications but are still charged to the user's account.

GLOBAL AND LOCAL RELOCATION FACTORS

A global relocation factor is maintained for referencing locations in virtual memory. Initially the global relocation factor is set to zero. The global relocation factor can be changed by the $SET command.

A local relocation factor, which overrides the global relocation factor, can be given in the $ALTER, $DISPLAY, $MODIFY, $RESTART and $START commands, and remains in effect for the duration of the command, unless changed by a subsequent local relocation factor in the same command. A displacement given in the same command is added to the current value of the relocation factor to provide an absolute virtual memory address. This computed absolute virtual memory address is used by the command to reference the storage location or block of locations desired.

MTS COMMANDS

The following notation conventions are used in the prototypes of the commands:

- lower case - represents a generic type which is to be replaced by an item supplied by the user.
- upper case - indicates material to be repeated verbatim in the command.
- brackets [ ] - indicates that material within the brackets is optional.
- braces {} - indicates that the material within the braces represents choices, from which exactly one must be selected. The choices are separated by vertical bars.
- dots ... - indicates that the preceding syntactic unit(s) may be repeated.
- underlining - indicates the minimum abbreviated form of the command or parameter. Longer abbreviations are accepted.

The following pages give a complete summary of the commands in the MTS command language.

Summary of MTS Command Prototypes

```
$ALTER location value ... ...

      location  GRx
                FRx
                [RF={hhhhhh|GRx}] xxxxxx
      value     {hhhh|X'hhhh'}
                C'xxxx'
                F'yyyy'
                H'yyyy'
```

```
$CALC {expression|decimal-number|hexadecimal-number}

$CANCEL [[*...*] [JOB] nnnnnn] [{CCID|ID}=xxxx]PW=yyyyyy]]

$COMMENT [text]

$CONTROL FDname control-command

$COPY [[FROM] FDname1] [[TO] FDname2]

$CREATE filename [SIZE={n|nP|nT}] [TYPE={LINE|SEQ|SEQWL}]

$DEBUG [objectFDname] [MAP[=mapFDname]] [NOMAP] [XREF] [I/OFDnames]
     [limits] [PAR=parameters]

$DESTROY filename [OK|O.K.|!]

$DISPLAY [ON FDname] [format] location ...

      format     {HEX|NOHEX}
                 {MNEM|NOMNEM}
                 {BCD|EBCD|NOBCD|NOEBCD}
                 {SP1|SSPC|SGLS|SP2|DSPC|DBLS}
                 {ORL=S|ORL=L}
      location   GRx
                 FRx
                 [RF={hhhhhh|GRx}] xxxxxx[...xxxxxx]
                 PSW
                 VMSIZE
                 {$|COST}
                 SIGFILE
                 {AFDNAME|AFD}

$DUMP [ON FDname] [format] ...

      format     {HEX|NOHEX}
                 {MNEM|NOMNEM}
                 {BCD|EBCD|NOBCD|NOEBCD}
                 {SP1|SSPC|SGLS|SP2|DSPC|DBLS}
                 {ORL=S|ORL=L}
                 {LIB|NOLIB}

$EDIT [{filename|:edit-command}]

$EMPTY filename [OK|O.K.|!]

$ERRORDUMP

$GET FDname

$HEXADD operand operand ...

      operand    {hhhh|GRx|RF}
```

```
$HEXSUB operand operand ...

      operand    {hhhh|GRx|RF}

$INQUIRE parameter [/parameter] ...

      parameter A device-type
                ACTIVE
                ALL
                B
                C [signon-id|task-number]
                CNTR
                CONFIG
                D device-name
                EXEC [LOCAL|RMTS|remote-station-id]
                H
                HELD|HOLD
                JOB  receipt-number
                L [S|MZ]
                LOCAL
                M
                ME
                MTS
                N
                O
                OS [signon-id|receipt-number|ALL]
                PAGES
                PLOT  [signon-id|receipt-number|ALL]
                PRINT [LOCAL|RMTS|remote-station-id]
                PRIORITY
                PUNCH [LOCAL|RMTS|remote-station-id]
                QUE [CNTR|EXEC|PRINT|PUNCH|ME|remote-station-id|
                  signon-id]
                receipt-number
                remote-station-id
                REMOTES|RMTS
                S [L]
                SAME
                SIGMSG
                STATUS
                STRANDS
                T device-type
                TAPES
                task-number
                U signon-id
                signon-id
                USERS
                *[receipt-number] [ME|signon-id]

$LIST [FDname1] [[ON] FDname2]

$LOAD [objectFDname] [MAP[=mapFDname]] [NOMAP] [XREF] [I/OFDnames]
     [limits] [PAR=parameters]
```

```
      limits    TIME={t|tS|tM}
                PAGES=p
                CARDS=c

$MODIFY location value ...  ...

$MOUNT [request [; request] ...  ]

$NET [*PDN*] [network-command ... ]

$NUMBER [[starting-number] [[,]increment]|[CONTINUE]

$PERMIT [filename [ALL|RUN|PRJNO|NONE|CNW|RO]]

$RELEASE [*pdn*]

$RESTART [[AT] location] [MAP[=mapFDname]] [NOMAP] [XREF] [I/OFDnames]
        [limits]

      limits    TIME={t|tS|tM}
                PAGES=p
                CARDS=c

$RUN [objectFDname] [MAP[=mapFDname]] [NOMAP] [XREF] [I/OFDnames]
    [limits] [PAR=parameters]

      limits    TIME={t|tS|tM}
                PAGES=p
                CARDS=c

$SDS [debug-command]

$SET keyword ...

      keyword   AFDECHO={ON|OFF}
                CASE={UC|LC|MX|MIXED}
                CMDSKP={ON|OFF}
                CONTCHAR=character
                COST={ON|OFF}
                CREAFD={ON|OFF}
                DEBUG={ON|OFF}
                DEVCHAR=character
                ECHO={ON|OFF}
                EDITAFD={ON|OFF}
                ENDFILE={ON|OFF|NEVER}
                ERRORDUMP={ON|OFF|FULL}
                FILECHAR=character
                IC={ON|OFF}
                LIBR={ON|OFF}
                LIBSRCH={OFF|FDname}
                LNS=character
                PFX={ON|OFF}
                PW=characters
```

```
                    RF={hhhhhh|GRx}
                    SCRFCHAR=character
                    SEQFCHK={ON|OFF}
                    SHFSEP=character
                    SIGFILE={OFF|FDname}
                    SIGFILEATTN={ON|OFF}
                    SYMTAB={ON|OFF}
                    TDR={ON|OFF}
                    TERSE={ON|OFF}
                    TRIM={ON|OFF}
                    UNLOAD={ON|OFF}
                    *LIBRARY={ON|OFF}
                    $={ON|OFF}

$SIGNOFF [SHORT|$]

$SIGNON ccid [keywords] ['d name']

      keywords  PW=password
                TIME={t|tS|tM}
                PAGES=p
                CARDS=c
                COPIES=n
                PRINT={QN|TN}
                ROUTE=station
                CROUTE=station
                PROUTE=station

$SINK {FDname|PREVIOUS}

$SOURCE {FDname|PREVIOUS}

$START [[AT] location] [MAP[=mapFDname]] [NOMAP] [XREF] [I/OFDnames]
      [limits]

      (see $RESTART command)

$UNLOAD [CLS=xxx]

$UNNUMBER
```

ALTER

COMMAND DESCRIPTION

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

Prototype: $ALTER location value ... ...

Each alteration requires a pair of the following parameters, the first specifying what is to be altered and the second specifying the new contents. Any number of items may be altered with a single ALTER command.

location

"location" is the general register, floating-point register, or virtual memory location(s) that is to be altered. "location" may be given in one of the following forms:

GRx

GRx specifies the general register "x", where "x" is a decimal integer from 0 to 15 or a hexadecimal integer from 0 to 9, A to F.

FRx

FRx specifies the floating-point register "x", where "x" is one of the integers 0,2,4, or 6.

[RF={hhhhhh|GRx}] xxxxxx

This specifies a virtual memory location given by an optional local relocation factor and a displacement. "hhhhhh" is the hexadecimal value of a local relocation factor or GRx indicates the general register whose contents are to be used as a local relocation factor. "xxxxxx" is the hexadecimal value of a displacement. The displacement is added to the current value of the relocation factor to provide an absolute 24-bit virtual memory address. If a local relocation factor is not specified, the global relocation factor is used. The global relocation factor is initially zero, but may be changed by the $SET command. When a local relocation factor is specified in the command, it remains in effect

for the remainder of the command unless subsequently overridden by a second local relocation factor specification.

value

The new contents are specified by any one of the following constant expressions:

hhhh or X'hhhh'

Any hexadecimal constant expression of any length may be given.

C'xxxx'

Any EBCDIC character expression of any length may be given between the delimiting primes; a prime in the character string must be represented by two consecutive primes.

F'yyyy' or H'yyyy'

A fullword (F) or halfword (H) decimal constant expression may be given consisting of a sign followed by decimal digits all enclosed in primes. The "+" sign is optional; the "-" sign is required. Decimal constants may not be specified for floating-point registers.

Description: The new constant given by the parameter "value" replaces the contents of the register or virtual memory location specified by "location". Register numbers and virtual memory addresses are checked for validity and a complaint is made if an illegal value is specified. At all times, a virtual memory address less than 500000 (hexadecimal) is illegal. If a privileged program is currently loaded, all locations are illegal. The parameter pairs in the command are processed from left to right.

General registers are altered as follows:

A character constant is truncated or padded with trailing blanks to four bytes (characters) and placed, left-justified, into the register.

The integer value of a hexadecimal constant (consisting of one to eight hexadecimal digits including leading zeros) is loaded, right-justified, into the register.

The integer value of a decimal constant is loaded into the register.

Floating-point registers are altered as follows:

A character constant is truncated or padded with trailing blanks to eight bytes (characters) and placed, left-justified, into the register.

A hexadecimal constant is truncated or padded with trailing zeros to eight bytes and placed, left-justified with leading zeros retained, into the register.

Virtual memory is altered as follows:

A character constant is placed, one character per byte into consecutive virtual memory locations.

A hexadecimal constant is placed, two hexadecimal digits per byte with leading zeros retained, in consecutive memory locations. If an odd number of hexadecimal digits is given, the last byte of memory altered has bits 4-7 set to zero.

The integer value of a decimal constant is loaded, without regard to boundary alignment, into either four or two bytes (for fullword or halfword constants) starting with the byte specified by "location".

Examples:

```
$ALTER GR3 1A3E0 FR6 X'41104'
```

The hexadecimal constant 0001A3E0 is placed in GR3 and the hexadecimal constant 4110400000000000 is placed in FR6.

```
$ALTER RF=5188E2 200 X'D502CC7E6000' 400 X'05EF' GRA 0
```

The hexadecimal constant D502CC7E6000 is placed in virtual memory location 518AE2, the hexadecimal constant 05EF is placed in location 518CE2, and the constant zero is placed in GR10.

```
$A RF=51A800 AEC F'-1000' RF=519600 2B6 C'DON''T DO IT'
```

The decimal constant -1000 is placed in virtual memory location 51B2EC, and the character constant DON'T DO IT is placed in location 5198B6.

CALC

COMMAND DESCRIPTION

Purpose: To perform arithmetic as a desk calculator in double precision, to evaluate arithmetic expressions containing decimal and/or hexadecimal numbers, and to convert decimal integers to hexadecimal numbers and vice versa.

Prototype: $CALC [{expression|number}]

where the parameter is an arithmetic expression when calculation is to be performed, or a single hexadecimal number or decimal integer when conversion is to be performed. Blanks may occur anywhere within the expression or numbers. If the $ operand is used in the expression, the result from the previous CALC command is used in its place.

Description: A decimal number may be signed and contain an explicit exponent; for example,

127 -2.2 +4 1e2 4.27e-8 .2,e+2

A hexadecimal number is a string of 1 to 8 hexadecimal digits (0-F) enclosed in single quote marks and optionally preceded by the letter X; for example,

X'503000' X'FF' 'FF' X'FFFFFFFF' '1B0'

When an arithmetic expression is the parameter, the expression is evaluated. The order of precedence is left to right, exponentiation first, followed by multiplication and division, then fcllowed by addition and substraction. Both decimal and hexadecimal numbers may be used in the expression. The permissible operators are:

\+ (addition)

\- (subtraction)

* (multiplication)

/ (division)

Parentheses may also be used to define the order of precedence. A single hexadecimal number may be enclosed in parentheses to define it as an expression.

If the parameter is a single hexadecimal number, it is converted to a decimal integer. If the parameter is a single decimal integer (no decimal point or exponent), it is converted to a hexadecimal integer.

Note that it is possible to interrupt an executing program, perform calculations, and then resume execution of the program via the $RESTART command.

Examples:

```
$CALC 22/7
$CALC X'16'/7
$CALC (2**(.5))*(1+1)
$CALC $*3.5
```

The above examples evaluate arithmetic expressions.

```
$CALC 255
$CALC -1
```

The above examples perform hexadecimal conversions.

```
$CALC 'FFFFFFFF'
$CALC X'FF'
```

The above examples perform decimal conversions.

```
$CALC $
```

The above example converts the result from the previous CALC command; if the result was in decimal, the conversion is in hexadecimal, and vice versa.

CANCEL

COMMAND DESCRIPTION

Purpose: To cancel *PRINT*, *PUNCH*, OR *BATCH* jobs submitted from a terminal, or regular batch jobs that the user has previously submitted.

Prototype: $CANCEL [*...*] [[JOB]nnnnnn] [{ID|CCID}=xxxx [PW=yyyyyy]]

where the parameters may be given in any order.

... nnnnnn

"*...*" is either *PRINT*, *PUNCH*, or *BATCH*, and "nnnnnn" is the 6-digit job receipt number. If the appropriate "*...*" has not been released, then the associated job is cancelled. If the "*...*" has been released to HASP, then the job receipt must also be specified. If both the "*...*" name and the receipt number are given, and the receipt number does not match the currently-opened "*...*" job, then it is assumed the user is referring to an earlier "*...*" job and the cancel information is passed to HASP. If both the "*...*" name and the job receipt number are given but the receipt number is that belonging to a different "*...*", an error comment is printed.

{ID|CCID}=xxxx [PW=yyyyyy]

"xxxx" is the signon-id associated with the job to be cancelled. This is required only if it is different from the ccid that the user is signed on under when he issues the CANCEL command. The password "yyyyyy" associated with the ccid must also be supplied. If it is not supplied the user is prompted for it.

Restrictions: An "*...*" job cannot be cancelled once processing has begun by HASP. Thus, *PRINT* can be cancelled only if it is awaiting printing, *PUNCH* if it is awaiting punching, and *BATCH* if it is awaiting execution. To find out where a job is in processing, issue the MTS $INQUIRE command.

Cancelling an *PRINT* or *PUNCH* job while it is still open causes the page-line or punch charges for the job to be rebated automatically. Once the job has been released to HASP, it can be cancelled (subject to the constraints listed earlier), but the charges are not rebated automatically.

Examples:

```
$CANCEL *PRINT*
```

The current *PRINT* job is cancelled.

```
$CANCEL 610399
```

The job with receipt number 610399 is cancelled.

```
$CANCEL *PUNCH* 603321
```

The *PUNCH* job with receipt number 603321 is cancelled.

April 1974

COMMENT

COMMAND DESCRIPTION

Purpose: To allow insertion of comments on output to the terminal or the printer.

Prototype: $COMMENT [text]

Description: This command is ignored by the system. As with all commands, it is echoed cn *SINK* and *MSINK* unless the command $SET ECHO=OFF has been given.

Example:

```
$COM THIS IS A COMMENT COMMAND
```

CONTROL

COMMAND DESCRIPTION

Purpose: To allow the execution of control operations on certain types of files and devices.

Prototype: $CONTROL FDname control-command

Description: The "FDname" parameter specifies the file or device on which the control operation specified by the "control-command" parameter is to be performed. The "control-command" parameter begins with the first non-blank character after the "FDname" parameter and continues for the remainder of the command.

Control commands may also be specified from user programs via the CONTROL subroutine which is described in Volume 3.

Only certain types of files and devices currently allow control operations to be performed. A list of the acceptable devices and their control commands follows. The codes in parentheses refer to the device type returned by the subroutine GDINFO (see MTS Volume 3).

Magnetic Tapes (9TP)

Control Command Parameter	Function
Positioning:	
REW	Rewind
FSR[n]	Forward space "n" records
BSR[n]	Backspace "n" records
FSF[n]	Forward space "n" files
BSF[n]	Backspace "n" files
POSN{*n*|*EOT*|name}	Position to nth file, end-of-tape, or data set name, respectively
Blocking:	
{FORMAT|FMT|RECFM} fmt[(size][,lrecl])] where "fmt" is {U|F|FB|FBS|V|VB|VS|VBS}	Specify blocking format and, optionally, block size and/or logical record length
{SIZE|BLKSIZE} n	Specify block size (18≤n≤32767)

LRECL n	Specify logical record length (1≤n≤32767)
BLK{ON|OFF}	Enable or disable blocking

Label Processing:

DSN [name]	Specify data set name for next new file written
LP{ON|OFF}	Enable or disable label processing

Error Recovery:

RETRY n	Specify read error retry count (0≤n≤15)

Miscellaneous:

WTM[n]	Write "n" tape-marks
MODE {800|1600}	Specify tape mode
PUSH	Push current tape parameters into stack
POP	Pop tape parameters from stack

Note that in the FSR, BSR, FSF, BSF, and WTM parameters, "n" must be in the range from 0 to 32767. If omitted, "n" defaults to 1. For a complete description of these parameters, see the "Magnetic Tape Users Guide" in MTS Volume 4.

Terminals (TTY, 2741)

Control Command Parameter	Function
BDCST[={ON|OFF}]	Enable or disable printing of messages from the operator
CC[={ON|OFF}]	Enable or disable carriage control
COL={ON|OFF}[;lm,rm]	Print only a specified column range in output lines
DCC[={x|ON|OFF}]	Redefine the device command character; enable or disable recognition of device commands
DLC[={x|ON|OFF}]	Redefine the delete-line character; enable or disable delete line function
DPC[={x|ON|OFF}]	Redefine the delete-previous character; enable or disable delete previous function

EFC[={x|ON|OFF}]	Redefine the end-of-file character; enable or disable end-of-file function
GOLF=ddd	Substitute typing elements for Selectric terminals
HEX[={x|ON|OFF}]	Enable or disable hexadecimal input editing; redefine the hexadecimal input delimiter
JOB	Retrieve line adapter and task numbers
K[={UC|LC}]	Specify alphabetic conversion mode for input lines
LEN={n|OFF}	Redefine the truncation length for output lines
{LMAR|RMAR}=n	Define position of left and right margin stops
LNC[={x|ON|OFF}]	Redefine the literal-next character; enable or disable literal next function
NOTE[={ON|OFF}]	Enable or disable "twitching" of type element
RESET	Reset all device command parameters
REV[={ON|OFF}]	Enable or disable byte-reversal function
{TABI|TABO}[={ON|OFF}[;x{t}[,t...]]]	Enable or disable logical tab stops; define logical tab character; establish positions of logical tab stops
TERSE[={ON|OFF}]	Enable or disable terse messages from the terminal device support routine
UCI[=ON|OFF}]	Enable or disable lowercase conversion for input lines
UCO[={ON|OFF}]	Enable or disable lowercase conversion for output lines
WARN={ON|OFF}	Enable or disable two minute warning on ten minute line

The default values for the above parameters vary depending on the type of terminal being used. In cases where the equal sign is optional, if it is omitted, the switch setting is reversed. For a complete description of these parameters, see the section "Memorex 1270 Device Support Commands" in MTS Volume 4.

MERIT Network (MNET)

Control Command Parameter	Function
ATTN	Send an attention interrupt to the remote host
BIN={ON|PFF}	Enable or disable binary translation for output
CC={ON|OFF|CHECKS}	Enable or disable logical carriage control
{CLEAR|RESET}	Resets all control parameters
DON'T	Prevent a connection from being released at signoff time
EOF	Send an end-of-file to the remote host
HEX={ON|OFF} ["x"]	Enable or disable hexadecimal translation
LEN=n	Specify length of received records
PFX={ON|OFF}	Enable or disable prefix processing
TAB={ON|OFF}["x"][t1 t2 ...]	Control processing of tab characters on input
UC={ON|OFF}	Enable or disable lowercase conversion

For a complete description of these parameters, see the section "MERIT Network Users Guide" in MTS Volume 4.

PRINT (HPTR), *PUNCH* (HPCH), *BATCH* (HBAT)

Control Command Parameter	Valid Pseudo-device Names
HOLD	*PRINT*, *PUNCH*, *BATCH*
RELEASE	*PRINT*, *PUNCH*, *BATCH*
CANCEL	*PRINT*, *PUNCH*, *BATCH*
ROUTE=station	*PRINT*, *PUNCH*
NAME='d name'	*PRINT*, *PUNCH*
PRINT={QN|TN}	*PRINT*
COPIES=n	*PRINT*
PROUTE=station	*PRINT*
CROUTE=station	*PUNCH*

For a description of these parameters, see the section "Batch Jobs From a Terminal" in this volume.

Examples:

```
$CONTROL *TAPE* REW
```

A REW control command is given the pseudo-device *TAPE* which is a 9-track magnetic tape.

```
$COPY *SOURCE* CNTRLFILE
$CONTROL *SINK* LEN=100
$RUN PROGRAM SCARDS=INPUTDATA SPRINT=*SINK*
$CONTROL *SINK* LEN=72
$ENDFILE
```

The file CNTRLFILE is established which can be used as a source file of commands for running a program on a Teletype. The CONTROL commands set the terminal output length to 100 characters for the duration of the program and then set the terminal output length to the default 72 characters. This sequence of commands can be started with the command $SOURCE CNTRLFILE.

COPY

COMMAND DESCRIPTION

Purpose: To copy from a file or device to another file or device.

Prototype: $COPY [[FROM] FDname1] [[TO] FDname2]

Two FDnames may be given as parameters:

FDname1

FDname1 specifies the file or device that contains the lines to be copied (the input). FDname1 may be an explicit concatenation of files and devices with line number ranges. If FDname1 is omitted, the input lines are read from the currently active file (*AFD*).

FDname2

FDname2 specifies the file or device that is to receive the copied lines (the output). FDname2 may be an explicit concatenation of files and devices with line number ranges. If FDname2 is omitted, the output lines are written on *SINK*.

FROM, TO

FROM and TO are optional "directive" words. If a FROM or TO is present, it specifies that the following parameter is FDname1 or FDname2, respectively. If FDname1 is omitted or follows FDname2, TO must precede FDname2 or FROM must precede FDname1 (or both). If FROM or TO are used in an ambiguous or unusual manner, the command is not executed and an error comment is produced.

Description: The $COPY command is a series of read and write operations. It causes lines to be read sequentially from FDname1 and written on FDname2 until the end of the file or the end of a line number range is encountered on FDname1.

For line files, the read operation uses the line numbers of FDname1. For sequential files and devices, the read operation simulates line numbers, although a beginning line number and increment may be specified for a device. For line files and devices, the write operation generates line numbers starting at 1 with an increment of 1 unless a beginning line number and increment is specified on FDname2. For sequential files, the write operation ignores line numbers and writes

the lines at the end of the file. If the @I modifier is used on FDname2, the write operation uses the line numbers from the read operation for generating line numbers for the write operation. Hence, if an exact copy of a line file is wanted (each line number in FDname2 having the same line number as in FDname1), the COPY command must be given in the form:

```
$COPY FDname1 FDname2@I
```

For line files (or sequential-with-line-numbers-files), lines may be written on FDname2 sequentially or indexed. If a line file is written sequentially, renumbering of lines occurs; however, the user can specify the beginning line number and increment for FDname2. For sequential files and devices, lines are written on FDname2 sequentially. If a line file is copied to a sequential file, the line numbers are lost. See the section "I/O Modifers" in the appendix to "Files and Devices" for a further description of the use of modifiers with read and write operations.

A complaint is made if either FDname1 or FDname2 does not exist, is not available, or is the wrong type (output or input, respectively).

Examples:

```
$COPY A TO B
```

File A is copied to file B. If B is a line file, new line numbers are generated for B starting at 1 and incrementing by 1. The line numbers from file A are not carried over to file B.

```
$COPY A B@I
```

File A is copied to file B. The line numbers from A are retained in B. If A is a sequential file, these line numbers start from 1 and increment by 1. File B must be a line file (or a sequential-with-line-number file).

```
$COPY A+B(5,20) C(10,,10)
```

File A and lines 5 through 20 of file B are copied to file C. The line numbers of C start at 10 and are incremented by 10.

```
$C A
```

File A is copied to *SINK* (default).

Comment: The following list of examples illustrates how the COPY command behaves. "x" and "y" stand for any parameter; "a" and "b" stand for any parameter that is not FROM or TO.

```
$COPY a              "a" is copied to *SINK*.
$COPY FROM x         "x" is copied to *SINK*.
$COPY TO y           *AFD* is copied to "y".
$COPY a b            "a" is copied to "b".
$COPY x TO y         "x" is copied to "y".
$COPY a FROM x       "x" is copied to "a".
$COPY FROM x a       "x" is copied to "a".
$COPY TO x a         "a" is copied to "x".
$COPY FROM x TO y    "x" is copied to "y".
$COPY TO x FROM y    "y" is copied to "x".
```

All other combinations of parameters are considered erroneous or ambiguous and cause an error comment to be produced. For example,

```
$COPY TO

$COPY FILE FROM
```

Note that if the current file name character (initially #) is prefixed to FROM or TO, they lose their special interpretation.

CREATE

COMMAND DESCRIPTION

Purpose: To create either a permanent or temporary file.

Prototype: $CREATE filename [keywords]

Only the "filename" parameter giving the name of the file to be created is required. The other legal keyword parameters that may be given are:

SIZE={n|nP|nT}

The SIZE keyword specifies the estimated size of the file to be created. The size may be given in one of three forms:

n - the number of 40 byte lines,
nP - the number of 4096 byte pages, or
nT - the number of 7294 byte tracks.

If the SIZE parameter is omitted, the default size of a permanent file is two pages or approximately 90 lines and the default size of a temporary file is 9 pages or approximately 700 lines, depending upon the format of the data. If this size is exceeded when the file is used, an attempt is made by the system to extend the file.

TYPE={LINE|SEQ|SEQWL}

The TYPE keyword specifies the type of file to be created. The type may be given in one of three forms:

LINE - a line file is created,
SEQ - a sequential file is created, or
SEQWL - a sequential file with line numbers is created.

If the TYPE keyword is omitted, the default type is LINE.

Description: The $CREATE command can be used to create either a permanent or temporary file. The parameter "filename" gives the name of the file to be created.

When the command has been given, MTS checks to make sure that a file of the given name does not already exist. A complaint is made if the file already exists. Then MTS checks the

user's file space allocation to determine if he has enough space remaining to allow creation of the file. Lastly, MTS attempts to acquire the space for him. If all three steps are successful, MTS informs the user of the successful creation of the file. The file is empty when created and becomes the currently active file in the system (unless the command $SET CREAFD=OFF has previously been issued). The pseudo-device *AFD* is associated with the file.

The SIZE parameter gives an approximation to the number of bytes that can actually be stored in the file. The actual capacity of the file is affected by the type of the file, the location of the file, and the length of the lines stored in the file.

Since temporary files are created automatically when first used as an FDname in a command or subroutine call, explicit creation of temporary files is necessary only when something other than the default specifications are needed.

Examples:

```
$CREATE A
```

The file A is created. Since there are no keyword parameters given, the file is a line file of default size, located on a disk.

```
$CR -B SIZE=10P TYPE=SEQWL
```

The temporary file -B is created. The file is sequential with line numbers and with a size of 10 pages.

DEBUG

COMMAND DESCRIPTION

Purpose: To load a program and enter into debug command mode.

Prototype: DEBUG [objectFDname] [MAP]=mapFDname]] [NOMAP] [XREF] [I/OFDnames] [limits] [PAR=parameters]

The following parameters may be given:

objectFDname

"objectFDname" specifies the file(s) or device(s) containing the program to be loaded and debugged. If omitted the program is loaded from *SOURCE*.

MAP[=mapFDname] [NOMAP] [XREF]

The MAP and XREF parameters are used to obtain a load map and cross reference listing from the dynamic loader. These are not normally needed when in debug mode. See the RUN command description for further details.

I/OFDnames

The keyword parameters "I/OFDnames" are the assignments of logical I/O units to files or devices for use by the loaded program during execution. The logical I/O unit assignments establish the I/O subroutines which will be used by the loaded program for input and output of data. Where no specifications are stated, the following default assignments occur:

```
SCARDS=*SOURCE*
SPRINT=*SINK*
SPUNCH=*PUNCH* (batch mode if global card est. > 0)
SERCOM=*MSINK*
GUSER=*MSOURCE*
```

The logical I/O units 0 through 19 have no default specifications. See the section "Files and Devices" in this volume and subroutine descriptions for SCARDS, SPRINT, SPUNCH, SERCOM, GUSER, READ, and WRITE in Volume 3 for further details on the use of these subroutines.

The logical I/O unit assignments may be initially assigned or reassigned in debug mode via the SET debug command.

FORTRAN users are reminded that MTS logical I/O units 0 through 19 are not necessarily the same as the FORTRAN logical I/O units 0 through 19. The FORTRAN I/O unit routines may default the FORTRAN logical I/O units independently of MTS. Moreover, if this has happened, the meaning of the units cannot be reassigned in debug mode.

limits

The keyword parameters "limits" specify local limits for time, pages printed, and cards punched. See the RUN command description for further details.

PAR=parameters

The PAR keyword specifies an arbitrary string of characters to be passed to the loaded program. This is usually a parameter list for the program and its interpretation depends on the loaded program. PAR keywords must be the last parameter field specified in the command. The parameter list is terminated by the end of the command line. Note that the parameter field always has a blank added after the last character and the count is incremented by one.

The parameter list may be initially assigned or reassigned in debug mode via the SET debug command.

Description: The DEBUG command calls upon the dynamic loader to load the object program into virtual memory. If there are unresolved external symbol references after loading from "objectFDname", loading continues from *LIBRARY (the system library). Only those parts of *LIBRARY required to resolve the references are loaded. If there are still unresolved external references, a fatal loading error exists. In conversational mode, the loader prompts for more loader input; in batch mode an error comment is produced and the loading terminates immediately. The search of the system library may be suppressed by the $SET LIBR=OFF command. SDS processes any symbol table information associated with the program.

If there were no fatal errors, control is transferred to debug command mode. In debug mode the user may use the facilities of SDS to display or modify parts of the loaded program and to initiate execution. SDS monitors the performance of the program. See the section "Debug Mode" for further details on the use of SDS.

The parameter string (specified by the PAR keyword) is passed as follows: GR1 contains the location of a fullword address constant which points to a region containing a halfword count (halfword aligned) followed by an EBCDIC character region (of

byte-length specified by the count) containing the parameter string. The left most bit of the address constant is 1.

If files or devices specified by "I/OFDnames" are non-existent or not available, 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 executed refers to the logical I/O unit, either the user is given the opportunity to respecify the FDname (in conversational mode) or execution is terminated (in batch mode).

Example: $DEBUG OBJPROG 5=INPUT 6=OUTPUT

This loads the program OBJPROG and transfers control to debug mode. Logical I/O units 5 and 6 are assigned to the files INPUT and OUTPUT respectively.

April 1974

DESTROY

COMMAND DESCRIPTION

Purpose: To destroy a private file or a temporary file.

Prototype: DESTROY filename [{OK|O.K.|!}]

Description: The "filename" parameter specifies the name of the file to be destroyed. A complaint is made if the parameter is missing, or if the file specified does not exist, or if it is a public file.

The destroyed file is deleted from the user's file catalog and the space occupied by the file is returned to the public domain. The user is informed when the file has been destroyed successfully.

If the user is at a terminal, confirmation is requested before a permanent file is destroyed. Confirmation is not requested for temporary files. The command may be confirmed by the response "OK", "O.K." or "!". Any other response causes the command to be cancelled. The confirmation may be given as the second parameter of the command.

Examples: $DESTROY A

This causes file "A" to be destroyed. The user is prompted for confirmation in conversational mode.

$DESTROY B OK

DISPLAY

COMMAND DESCRIPTION

Purpose: To display the contents of general registers, floating-point registers, the program status word, the user's virtual memory size, the accumulated cost of the current job, and/or specified virtual memory location(s).

Prototype: $DISPLAY [ON FDname] [format] location ...

"location" is the only required parameter. As many "location" and "format" parameters may be given as desired. The only restriction on the order of parameters in the command line is that "ON FDname" must appear first if it appears at all. "format" parameters affect only the "location" parameters which follow it.

ON FDname

"FDname" is the file or device to which the output from the $DISPLAY command is written. If "FDname" is omitted, the output is written on *SINK*. A complaint is made if "FDname" specifies a non-existent or unavailable file or device.

format

The format of the display may be specified by any combination of the following option switches:

HEX	hexadecimal conversion
NOHEX	hexadecimal conversion off
MNEM	mnemonic and hexadecimal conversion
NOMNEM	mnemonic and hexadecimal conversion off
BCD	EBCDIC conversion
EBCD	EBCDIC conversion
NOBCD	EBCDIC conversion off
NOEBCD	EBCDIC conversion off
SP1	single-spacing
SSPC	single-spacing
SGLS	single-spacing
SP2	double-spacing
DSPC	double-spacing
DBLS	double-spacing

```
ORL=S  short output record (70 characters)
ORL=L  long output record (130 characters)
```

If not specified, the following default option settings apply:

```
NOHEX
NOMNEM
NOEBCD
SP1
ORL=L
```

If the NOHEX, NOMNEM and NOEBCD are all specified simultaneously (explicitly or by default), the output is with hexadecimal conversion.

"format" parameters affect only "location" parameters that follow the "format" parameters and are blocks of virtual memory. They do not affect single memory locations or other items.

location

"location" specifies what is to be displayed. This may be any of the following:

GRx

GRx specifies the general register "x", where "x" is a decimal integer from 0 to 15 or a hexadecimal integer from 0 to 9, A to F, or "S" if all general registers are to be displayed.

FRx

FRx specifies the floating-point register "x", where "x" is one of the integers 0,2,4, or 6, or "S" if all floating-point registers are to be displayed.

[RF={hhhhhh|GRx}] xxxxxx[...xxxxxx]

This specifies a virtual memory location or range of locations given by an optional local relocation factor and a displacement or range of displacements. "hhhhhh" is the hexadecimal value of a local relocation factor or GRx indicates the general register whose contents are to be used as a local relocation factor. "xxxxxx" is the hexadecimal value of a displacement. A range of displacements can be given by "xxxxxx...xxxxxx". The displacement is

added to the current value of the relocation factor to provide an absolute 24-bit virtual memory address. If a local relocation factor is not specified, the global relocation factor is used. The global relocation factor is initially zero, but may be changed by the $SET command. When a relocation factor is specified in the command, it remains in effect for the remainder of the command unless subsequently overridden by a second local relocation factor specification.

PSW

PSW specifies the program status word at the time the last loaded program terminated.

VMSIZE

VMSIZE specifies the current size of the user's virtual memory in a decimal number of pages.

$ or COST

Either $ or COST specifies the accumulated cost of the current job. This includes all charges up to the current time except permanent file storage charges and charges for tapes still mounted.

SIGFILE

SIGFILE displays the current and new (if any) SIGNON files.

AFDNAME or AFD

AFDNAME or AFD displays the name of the currently-active file or device.

Description: The $DISPLAY command displays general registers, floating-point registers, the program status word, the user's virtual memory size, the accumulated cost of the current job, and/or specified virtual memory locations.

The general registers, floating-point registers and the PSW are displayed in labeled hexadecimal format.

Blocks of virtual memory are displayed in hexadecimal, mnemonic and hexadecimal, and/or EBCDIC format.

Whenever a "location" parameter is encountered in a $DISPLAY command line, it is processed immediately using the "format" parameters and relocation factor in effect at that time. "format and "location" parameters are processed from left to right in the command line.

One ambiguity may occur. EBCD can represent either the EBCD option setting or a hexadecimal address displacement. It is interpreted as the EBCD option setting. To display the single location EBCD, 0EBCD must be specified.

Examples:

```
$DISPLAY GR3 FRS EBCD 518E08...518FA6
```

This displays GR3 and all the floating-point registers on *SINK* in single-spaced hexadecimal format, and displays virtual memory locations 518E08 through 518FA6 (assuming a global relocation factor of zero) in single-spaced EBCDIC format.

```
$D ON DISPLAYFILE ORL=L GRS PSW VMSIZE
```

This displays on the file DISPLAYFILE all the general registers, the program status word, and the size of the user's virtual memory in long record hexadecimal format.

```
$D MNEM EBCD SP2 RF=518000 200...480 NOMNEM NOEBCD 800...A80
```

This displays virtual memory locations 518200 through 518480 on *SINK* in double-spaced mnenonic, hexadecimal, and EBCDIC conversion format; and displays locations 518800 through 518A80 in double-spaced hexadecimal format.

DUMP

COMMAND DESCRIPTION

Purpose: To display the contents of general registers, floating-point registers, the program status word, and the virtual memory locations associated with the user's current loaded program.

Prototype: $DUMP [ON FDname] [format] ...

As many "format" parameters may be given as desired. The only restriction on the order of parameters in the command line is that "ON FDname" must appear first if it appears at all.

ON FDname

"FDname" is the file or device to which the output from the DUMP command is written. If "FDname" is omitted, the output is written on *SINK*. A complaint is made if "FDname" specifies a non-existent or unavailable file or device.

format

The format of the display may be specified by any combination of the following option switches:

HEX	hexadecimal conversion
NOHEX	hexadecimal conversion off
MNEM	mnemonic and hexadecimal conversion
NOMNEM	mnemonic and hexadecimal conversion off
BCD	EBCDIC conversion
EBCD	EBCDIC conversion
NOBCD	EBCDIC conversion off
NOEBCD	EBCDIC conversion off
SP1	single-spacing
SSPC	single-spacing
SGLS	single-spacing
SP2	double-spacing
DSPC	double-spacing
DBLS	double-spacing
ORL=S	short output record (70 characters)
ORL=L	long output record (130 characters)

LIB dump all storage including library space
NOLIB dump only non-library space

If not specified, the following default option settings apply:

```
NOHEX
NCMNEM
NOEBCD
SP1
ORL=L
LIB
```

If the NOHEX, NOMNEM and NOEBCD are all specified simultaneously (explicitly or by default), the output is with hexadecimal conversion. If ORL=L is specified or defaulted, then both hexadecimal and EBCD conversion is given side by side.

Description: The $DUMP command displays the general registers, floating-point registers, the program status word, and the virtual memory locations associated with the user's current loaded program.

The general registers and floating-point registers are displayed in labeled hexadecimal format.

Blocks of virtual memory are displayed in hexadecimal, mnemonic and hexadecimal, and/or EBCDIC format.

Note that the NOLIB parameter does not work correctly unless the command "$SET SYMTAB=ON" is used before loading the program.

Examples: $DUMP

The general registers, floating-point registers, the program status word, and the virtual memory locations are displayed in single-spaced hexadecimal format on *SINK*.

$DU HEX EBCD SP2

The general registers, floating-point registers, the program status word, and the virtual memory locations are displayed in double-spaced hexadecimal and EBCDIC format.

EDIT

COMMAND DESCRIPTION

Purpose: To enter edit mode, invoking the context editor for making changes to a file.

Prototype: $EDIT [{filename|:edit-command}]

The legal parameters are:

filename

"filename" is the name of a line file to be edited. The editor cannot be used on sequential files. See the "Edit Mode" section for a description of the context editor and the editor command language.

edit-command

"edit-command" is any single edit mode command. It must be preceded by a colon ":".

Description: If "filename" is specified, and if EDITAFD is ON (see the $SET command), "filename" becomes the currently active file (*AFD*). If "filename" is omitted, or if ":edit-command" is specified, the following procedure is used to determine the file to be edited.

1) If the editor has not previously been used, or if the editor was last terminated by the edit mode command "STOP", the user is prompted for the file name.

2) If the editor was last terminated by the edit mode command "MTS", and

   a) EDITAFD is OFF, the file last specified on a $EDIT command is used. If this file is no longer available, an error message is produced.

   b) EDITAFD is ON, the currently active file (*AFD*) is used. If there is no currently active file, the user is prompted for the file name. If no parameter is given, the user is prompted to enter parameters until the parameter "MTS" is given.

If any of the above error conditions exist, the user is prompted to enter a file name.

Example: $EDIT DATAFILE

The context editor is invoked to edit the line file DATAFIle.

```
$EDIT :CHANGE 10 'A'B'
```

This command changes the first occurrence of the character A in line 10 of the currently active file to the character B and then returns to MTS command mode. It must be preceded by a ":".

EMPTY

COMMAND DESCRIPTION

Purpose: To empty the contents of a file without destroying the file.

Prototype: $EMPTY filename [{OK|O.K.|!}]

Description: The "filename" parameter specifies the name of the file to be emptied. A complaint is made if the parameter is missing, if the file specified does not exist, if the file is a public file, or if the file is a read-only file.

The current contents of the file "filename" are discarded. The user is informed when the file has been emptied successfully. The space occupied by the file is not released, a catalog entry still exists for the file, and the user continues to be charged for it.

If the user is at a terminal, confirmation is requested before a permanent file is emptied. Confirmation is not requested for scratch files. The command may be confirmed by the response "OK", "O.K.", or "!". Any other response causes the command to be cancelled. The confirmation may be given as the second parameter of the command.

Examples:

```
$EMPTY A

$EMPTY B OK
```

April 1974

ERRORDUMP

COMMAND DESCRIPTION

Purpose: To allow automatic program dumps in batch mode in case of an abnormal program termination.

Prototype: $ERRORDUMP

Description: If an executing program terminates abnormally, a dump of the registers and program storage region is given. Common abnormal terminations are a program interrupt, a call to the subroutine ERROR, and exceeding global or local time, page, or card estimates. This command is equivalent to the command $SET ERRORDUMP=ON.

This command has no effect in terminal mode since dumps are never automatically produced. Terminal users may use the DUMP, DISPLAY, or debug mode DISPLAY commands to inspect all or parts of virtual memory. A symbolic program dump may be obtained via the symbolic debugging system facilities in a similar manner. For details, see the "Debug Mode" section in this volume.

Example: $ER

GET

COMMAND DESCRIPTION

Purpose: To establish a file or device as the currently active file.

Prototype: $GET FDname

Description: The "FDname" parameter specifies the name of the file or device to become the currently active file. The file or device is opened and the pseudo-device *AFD* is associated with that file or device. A complaint is made if the parameter is omitted, if the file or device does not exist, if the file is a read-only file, or if the device is an input device. The previously active file is released even if one of these error situations exists. The line number for the $NUMBER CONTINUE command is reset to one.

Example: $GET A

April 1974

HEXADD

COMMAND DESCRIPTION

Purpose: To perform hexadecimal addition in command mode.

Prototype: $HEXADD operand operand ...

One or more operands separated with intervening blanks are given with the command. Each operand may be given in one of the three following forms:

hhhh	A hexadecimal number to be used as an operand.
GRx	A general register whose contents is to be used as an operand.
RF	The current global relocation factor.

Description: The hexadecimal numbers or the contents of the registers specified by the operands are added. Overflows are ignored.

The results of the HEXADD command appear in the form:

```
SUM = xxxxxxxx
```

Example:

```
#hexadd 1a2 2e81d
#SUM = 2E9BF
```

In this terminal mode example, the hexadecimal numbers 1A2 and 2E81D are added together to produce the sum 2E9BF.

HEXSUB

COMMAND DESCRIPTION

Purpose: To perform hexadecimal subtraction in command mode.

Prototype: $HEXSUB operand operand ...

One or more operands separated with intervening blanks are given with the command. Each operand may be given in one of the three following forms:

hhhh — A hexadecimal number to be used as an operand.

GRx — A general register whose contents is to be used as an operand.

RF — The current global relocation factor.

Description: The second and all following operands are subtracted from the first operand. Negative results are given with a minus sign preceding the absolute value of the difference.

The results of the HEXSUB command appear in the form:

```
DIFF = xxxxxxxx
```

Examples:

```
#hexs 2e9bf 1a2
#DIFF = 2E81D
```

In this terminal example, the hexadecimal number 1A2 is subtracted from 2E9BF to produce the difference 2E81D.

```
#hexs 1e57e 1a2 5bc
#DIFF = 1DE21
```

In this example, 1a2 is subtracted from 1e57e, and 5bc is subtracted from this result.

INQUIRE

COMMAND DESCRIPTION

Purpose: To display the current status of a batch job, an execution queue, a print queue, a punch queue, plotting jobs, OS batch jobs, UMMPS jobs, or other system activity.

Prototype: $INQUIRE parameter [/parameter] ...

The output for this command falls into five categories discussed after the list of parameters. One of the following parameters must be given. The minimum abbreviation (if any) of each parameter is underlined.

Batch Jobs

ACTIVE
: Information about all the active batch jobs is printed in Output Type 1 format.

B
: Information about all MTS batch tasks is printed in Output Type 4 format.

{HELD|HOLD}
: Information about all batch jobs that are held is printed in Output Type 1 format.

JOB receipt-number
: Information on the job with the specified receipt number is printed in Output Type 1 format. This form should be used if the receipt number is less than 5 digits in length.

ME
: Information about all jobs submitted by the signon id signed on is printed in Output Type 1 format. All jobs not yet completed or completed in the last 96 hours are listed.

OS [signon-id|six-digit-receipt-number|ALL]
: Information about the OS batch queue is printed in Output Type 1 format. If any batch job is found that fits the search criterion, it is displayed. Position in the queue gives the initiator class for the job. The last line of output prints the total number of jobs and records currently in the OS batch queue.

Search criterion can be

1) a particular signon id
2) a particular receipt number
3) all jobs
4) no jobs (no second operand); just gives summary.

PLOT [ALL|receipt-number|signon-id]
Information about the PLOT queue is printed in Output Type 1 format. If a plot job is found that fits the search criterion, it is displayed. The last line of output gives the total number of jobs and time to plot the jobs queued.

Search criterion can be:

1) a particular signon id
2) a particular receipt number (does not include destination code)
3) all jobs
4) no jobs (no second operand); just gives summary.

receipt-number
Information about the job with the specified receipt number is printed in Output Type 1 format.

SAME
Information about the job last asked about via the receipt number is printed in Output Type 1 format.

signon-id
Information about all jobs submitted under the specified signon id is printed in Output Type 1 format. Both unfinished jobs and jobs finished within the last 96 hours are included.

*[receipt-number] [ME|signon-id]
This parameter prints information about all jobs not yet completed by the signon id who is making the request (if the second operand is omitted or ME was specified), or information about jobs submitted by the signon id specified. In addition, all completed jobs belonging to the user that have completed whose receipt number is greater than or equal to the receipt number specified are indicated. If a receipt number is not specified, the starting point is 600000 which is the lowest *BATCH*, *PRINT* or *PUNCH* receipt number.

Queues

ALL
A summary is printed in Output Type 3 format of each queue for all jobs in the batch queue.

CNTR
A summary of each queue counting only jobs with output routed to the Computing Center is printed in Output Type 3 format.

EXEC [LOCAL|RMTS|remote-station-id]
The execution queue summary is printed in Output Type 3 format for the following second operands:

1) jobs routed to the center if CNTR or LOCAL is specified
2) jobs routed to remote stations if RMTS is specified
3) jobs routed to the specified remote station
4) all jobs if no second operand is specified.

LOCAL
A summary of each queue counting only jobs with output routed to the Computing Center is printed in Output Type 3 format.

PRINT [LOCAL|RMTS|remote-station-id]
The print queue summary is printed in Output Type 3 format for the following second operands:

1) jobs to be printed at the center if CNTR or LOCAL is specified
2) jobs routed to remote stations if RMTS is specified
3) jobs routed to the specified remote station
4) all jobs if no second operand is specified.

PUNCH [LOCAL|RMTS|remote-station-id]
The punch queue summary is printed in Output Type 3 format for the following second operands:

1) jobs to be printed at the center if CNTR or LOCAL is specified
2) jobs routed to remote stations if RMTS is specified
3) jobs routed to the specified remote station
4) all jobs if no second operand is specified.

QUE
The QUE parameter prints information in Output Type 2 and 3 formats for the following second operands:

null — Gives limited information about all jobs not yet completed in the batch queue and a queue summary. The information printed by the USERS parameter is first printed.

PRINT — Gives limited information about all jobs not yet completed in the PRINT queue and a queue summary.

PUNCH — Gives limited information about all jobs not yet completed in the punch queue and a queue summary.

EXEC — Gives limited information about all jobs not yet completed in the execute queue and a queue summary.

CNTR — Gives limited information about all jobs not yet completed whose output is routed to the Computing Center and a queue summary.

remote-station-id — Gives limited information about all jobs not yet completed whose output is routed to the specific remote station and a queue summary.

signon-id — Gives limited information about all jobs not yet completed that were submitted by the specified signon id and does not give a queue summary.

ME — Gives limited information about all jobs submitted by the signon id that is making the inquiry and have not yet been completed.

Note:
1) Any combination of PRINT, PUNCH and EXEC is valid for a single QUE command.
2) Only one (the last found) remote station id is valid for a single QUE command.
3) Only one (the last found) signon id is valid for a single QUE command.
4) Type of queue, location and signon id may appear in any combination in a single QUE command.

remote-station-id
A summary of each queue counting only jobs routed to that remote station is printed in Output Type 3 format.

{REMOTES|RMTS}
A summary of each queue counting only jobs routed to remote stations is printed in Output Type 3 format.

Devices

A device-type
This parameter prints a list of available devices of the type specified.

CONFIG
The number of storage modules (SSU), channel controllers (CCU), central processors (CPU), readers, printers, tape drives, and punches attached to the system is printed.

D device-name
Information about the task that owns the device specified is printed in Output Type 4 format.

L [S|MZ]
This parameter produces a list of started phone lines in one of three ways:

1) If second operand is null, a list of all phone lines started.
2) If second operand is S, a list of all phone lines started and the signon id of each user on each line.
3) If second operand is MZ, a list of only mezzanine phone lines.

T device-type
This parameter indicates what units are available for usage by tasks. All units offline are indicated and information about all tasks owning a device of the specified type is given.

TAPES
This is an alternate form of the command A 9TP.

Tasks

C [task-number|signon-id]
The task number, signon id, and CPU time used by the current user is printed for MTS tasks selected by one of the following second operands:

1) task number
2) signon id
3) batch jobs (if no second operand).

H
Information about all HASP and HASPLING tasks is printed in Output Type 4 format.

M
Information about all active MTS tasks is printed in Output Type 4 format.

N
Information about all active non-MTS tasks is printed in Output Type 4 format.

O
Information about all operator console tasks is printed in Output Type 4 format.

S [L]
This parameter prints a list of all users currently signed on.

task-number
Information about the task whose number is specified is printed in Output Type 4 format. This number can only be between the numbers 1 and 9999. If the number specified has more than 4 digits it is taken to be a receipt number.

U signon-id
This parameter prints information about the task owned by the specified signon id.

System

PAGES
This parameter lists the signon ids of all users currently connected to the system and the current number of pages of virtual memory the user has acquired. If the signon id and the number of pages is separated by a dash (-), the task is conversational; if separated by a colon (:), the task is batch.

PRIORITY
This parameter prints the table used to determine the priority for executing and printing batch jobs. For execution of jobs, only the estimated execution time is used to determine the priority. For printing of jobs, the actual number of pages to print is used. For a given priority, the maximum execution time and number of printed pages are given. The table produced by this command supersedes all tables published in any Computing Center documentation.

SIGMSG
This causes the current signon message to be printed, along with the mesage's length. This is useful when the signon message exceeds the terminal's default length and

the user would like to know what the entire message is. A %LEN command should be issued before issuing the INQUIRE command.

STATUS
Information about the status of the system is printed. This includes the mode that the system is in (attended or unattended, all batch execution held or released), the number of users signed on (see the USERS parameter below), and the batch backlog (see the ALL command).

STRANDS
Information about batch processors is printed. The output contains two types of entries. The form:

JJ-PP-SS

is common to both types of entries.

JJ is the maximum number of active MTS jobs that allow this processor to execute a batch job.

PP is the lowest priority job that this processor executes.

SS is the first priority level to be searched for an available batch job.

Processors not only execute a job, but also act as a transport system between MTS and HASP for batch input and printed and punched output.

JJ-PP-SS:TT - receipt-number signon-id

This form is used when a processor is active. It contains the information mentioned above along with the job number and the user being serviced. The type of function being accomplished by the processor is

TT = EX if a batch job is being executed;

BA if a batch job is being read in via *BATCH*;

PR if a listing is being queued for printing via *PRINT*;

PU if a deck of cards is being queued for punching via *PUNCH*.

USERS
This parameter prints the number of MTS tasks, conversational users, batch users, idle lines, pages of virtual

memory, and the number of pages of real memory being used. Also the percent of used HASP secondary storage is given. When this percentage is very high (over 85%), users should refrain from using the *PRINT*, *BATCH*, and *PUNCH* facilities.

Other

MTS

This parameter returns to command mode.

Description: Five types of output are available from the $INQUIRE command. Four types apply to several parameters, and are discussed here; the last type is parameter specific and is discussed with the specific parameter.

Type 1. Gives detailed information about a batch job's status, as it currently appears in one of the three batch queues (execute, print or punch), or as it appears in a file of finished jobs. The output takes one of the following forms:

1) JOB receipt-number signon-id (location) IS HELD IN function QUEUE, BECAUSE reason

This message indicates the job was held for the reason given.

2) JOB receipt-number signon-id (priority) IS function (location)

This job is currently being printed or punched at the location specified.

3) JOB receipt-number signon-id (priority) EXECUTION TASK number USED number SEC.

This job is currently executing as the task number given, and has used the indicated number of seconds.

4) JOB receipt-number signon-id HAS number PAGES AT LOC. location (delivery-code) date-finished time-finished

This job has finished all its scheduled functions on the date and time specified. The number of pages printed and punched output (if the word CARDS occurs after the time finished) is waiting for the user at the remote station or the delivery location indicated.

Please note, the time finished indicates when the computer has finished processing the job. This does not

reflect in any way the time it takes to separate, deliver, or distribute output, which must be done by humans. Also, jobs with large volumes of output routed to remote stations (as well as shorter jobs when large remote printing backlogs exist) are printed at the Computing Center and delivered to the remote stations. Thus the remote station and delivery code only indicate where the output eventually comes to rest. Production Control should be contacted about any confusion over remote stations and delivery codes.

5) JOB receipt-number signon-id (priority) IS IN POSITION number ON THE function QUEUE location

WAITING FOR: number (priority) [number (priority)...]

The job is waiting for the service indicated. Its number in line is given. If there are jobs scheduled ahead of it, the number of jobs at each priority level ahead of the specified job are listed on a second line.

Type 2. Gives abbreviated information about several jobs on a single print line. The number of jobs per line depends upon the device's line length.

signon-id receipt-number priority location-indicator type position

The signon id, receipt number, priority, and position of every job still requiring processing is listed.

The codes for type are:

EXEC - actively executing
EX - queued for execution
PRINT - actively printing
PR - queued for printing
PUNCH - actively punching
PU - queued for punching
HELD - currently held for some reason

The location indicator code is blank for local and '>' for remote jobs.

Type 3. Gives a summary of all jobs currently in a particular queue.

type QUEUE (location:amount) number ACTIVE number QUEUED: number (priority) number (priority)... number (HELD)

"type" is EXECUTE, PRINT or PUNCH.

"location" indicates what locations within the queue are included.

"amount"

1) is null or indicates EXECUTION IS HELD in the EXECUTE QUEUE.
2) indicates the total number of cards to be punched for all queued jobs in the PUNCH QUEUE.
3) is the number of QN and TN print train pages to be printed for all queued jobs in the PRINT QUEUE.

The number of active and queued jobs are indicated. For each non-empty priority level, the number of jobs at that priority are given. The number of jobs that are held are indicated.

Type 4. Gives information about a particular task.

task-number number-of-virtual-pages receipt-number job-table-address parameters devices-owned

The number of virtual pages allocated by the task is present only for virtual tasks. The parameters for MTS include signon id, project number, and receipt number if a batch job.

Examples:

```
$I ME
```

This prints out information about all the user's finished and unfinished jobs occurring in the last 96 hours.

```
$I
609839
SAME
```

INQUIRE stores the receipt number of the last asked about job for the current signon session. This sequence is useful if it becomes important for a user to determine when a particular job is finished.

```
$I USERS
```

This prints out information concerning the present system load.

LIST

COMMAND DESCRIPTION

Purpose: To list a file or device onto another file or device with line numbers.

Prototype: $LIST [FDname1] [[ON] FDname2]

Two FDnames may be given as parameters:

FDname1

FDname1 specifies the file or device that contains the lines to be listed (the input). FDname1 may be an explicit concatenation of files and devices with line number ranges. If FDname1 is omitted, the input lines are read from the currently active file (*AFD*).

FDname2

FDname2 specifies the file or device that is to receive the listed lines (the output). FDname2 may be an explicit concatenation of files and devices with line number ranges. If FDname2 is omitted, the output lines are written on *SINK*.

ON

ON is an optional "directive" word. If FDname1 is omitted or follows FDname2, ON must precede FDname2.

Description: The LIST command is a series of read and write operations. It causes lines to be read sequentially from FDname1 and written on FDname2 until the end of the file or the end of a line number range is encountered on FDname1. The line number returned from the read operation on FDname1 is converted to a 12 character EBCDIC number and appended on the front of each line. This extended line is written on FDname2.

The LIST command uses line numbers and modifiers for the read and write operations in the same manner as in the COPY command.

A complaint is made if either FDname1 or FDname2 does not exist, is not available, or is the wrong type (output or input, respectively).

Examples: $LIST A

File A is listed on *SINK* (default). If A is a line file, the line numbers from file A are appended to output lines before they are written on *SINK*. If file A is a sequential file, line numbers starting at 1 and incrementing by 1 are appended to the output lines.

$LIST A(5,20) B

Lines 5 through 20 of file A are listed on file B. The line numbers from A are appended as above.

$L

The currently active file or device *AFD* (default) is listed on *SINK* (default) with line numbers appended as above.

Comments: The following list of examples illustrates how the LIST command behaves. "x" and "y" stand for any parameter; "a" and "b" stand for any parameter that is not ON.

$LIST a	"a" is listed on *SINK*.
$LIST ON x	*AFD* is listed on "x".
$LIST a ON b	"a" is listed on "b".
$LIST a b	"a" is listed on "b".
$LIST ON a x	"x" is listed on "a".

All other combinations are considered erroneous or ambiguous and cause an error comment to be produced. For example,

$LIST ON

Note that if the current file name character (initially #) is prefixed to ON, it loses its special interpretation.

LOAD

COMMAND DESCRIPTION

Purpose: To load a program without initiating execution.

Prototype: $LOAD [objectFDname] [MAP[=mapFDname]] [NOMAP] [XREF]
[I/OFDnames] [limits] [PAR=parameters]

The following parameters may be given:

objectFDname

"objectFDname" specifies the file(s) or device(s) containing the program(s) to be loaded. If omitted, the program is loaded from *SOURCE*.

MAP[=mapFDname] [NOMAP] [XREF]

The MAP keyword specifies the file or device "mapFDname" on which the loader is to write the load map. If the MAP keyword parameter is omitted, no load map is written. If the MAP keyword is given in the form of MAP, "mapFDname" defaults to *SINK*. The NOMAP parameter suppresses the printing of the load map. NOMAP need be given only if a previous MAP specification was made.

The XREF parameter specifies that a cross reference listing of external symbols occurring in the loaded programs is to be produced in addition to the load map. If XREF is given without the MAP keyword, MAP=*SINK* is assumed. If XREF is given with NOMAP, only the cross reference is produced.

I/OFDnames

The keyword parameters "I/OFDnames" are the assignments of logical I/O units to files or devices for use by the loaded program during execution. The logical I/O unit assignments establish the I/O subroutines which are used by the loaded program for the input and output of data. Where no specifications are stated, the following default assignments occur:

```
SCARDS=*SOURCE*
SPRINT=*SINK*
SPUNCH=*PUNCH* (batch mode if global card est. >0)
SERCOM=*MSINK*
GUSER=*MSOURCE*
```

The logical I/O units 0 through 19 have no default specifications. See the section "Files and Devices" in this volume and the subroutine descriptions for SCARDS, SPRINT, SPUNCH, SERCOM, GUSER, READ and WRITE in Volume 3 for further details on the use of these subroutines.

FORTRAN users are reminded that MTS logical I/O units 0 through 19 are not necessarily the same as the FORTRAN logical I/O units 0 through 19. The FORTRAN I/O routines may default the FORTRAN logical I/O units independently of MTS. Moreover, if this has happened, the meaning of these units cannot be reassigned by a RESTART command.

limits

The keyword parameters "limits" specify local limits for CPU time, pages printed and cards punched. These can be given in the form:

TIME={t|tS|tM}
PAGES=p
CARDS=c

These local limits can be used in both batch and terminal mode and are effective only for the execution of the program loaded by the LOAD command. See the description of the similar global limits in the $SIGNON command description for further details of limit specifications.

PAR=parameters

The PAR keyword specifies an arbitrary string of characters to be passed to the loaded program on initiation of execution. This is usually a parameter list for the program and its interpretation depends on the loaded program. The PAR keyword must be the last parameter field specified in the command. The parameter list is terminated by the end of the command line.

Description: The LOAD command calls upon the dynamic loader to load the object program into virtual memory. If there are unresolved external symbol references after loading from "objectFDname", loading continues from *LIBRARY (the system library). Only those parts of *LIBRARY required to resolve the references are loaded. If there are still unresolved external references, a fatal loading error exists. In terminal mode, the loader prompts for more loader input; in batch mode, an error comment is produced and the loading terminates immediately. The search of the system library may be suppressed by the $SET LIBR=OFF command. If $SET DEBUG=ON has been

specified, SDS processes only symbol table information associated with the loaded program.

After the program is loaded, control is returned to the user in command mode. The user can then use SDS or the commands $DISPLAY and $ALTER to display and modify parts of the loaded program. Execution of the program can be initiated by the $START command or by the debug mode RUN command.

The parameter string (specified by the PAR keyword) is passed as follows: GR1 contains the location of a fullword address constant which points to a region containing a halfword count (halfword aligned) followed by an EBCDIC character region (of byte-length specified by the count) containing the parameter string. The leftmost bit of the address constant is 1.

If files or devices specifed by "I/OFDnames" are non-existent or not available, the logical I/O unit referring to the unavailable file or device is set up such that the first time the program being executed refers to that logical I/O unit, either the user is given an opportunity to respecify the FDname (in terminal mode) or execution is terminated (in batch mode).

Example:

```
$LO OBJPROG+PROGLIB MAP 5=INPUT 6=OUTPUT
```

This loads the program and a private library in OBJPROG+PROGLIB. A load map is printed on *SINK*. Logical I/O units 5 and 6 are specified for the input and output of data during a later execution of the program which can be initiated by the START command or by the debug mode RUN command.

MODIFY

COMMAND DESCRIPTION

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

Prototype: $MODIFY location value ... ...

This command is identical to the $ALTER command.

MOUNT

COMMAND DESCRIPTION

Purpose: To mount magnetic tapes or MERIT network connections.

Prototype: $MOUNT [request [; request] ...]

request

"request" is the mount request for the item to be mounted. One or more requests separated by semicolons may be entered directly on the $MOUNT command. If the requests are omitted from the command line, they must be entered as separate input lines following the command (read from *SOURCE*) until terminated by an end-of-file.

The form of the request for mounting magnetic tapes is:

racknumber [ON] 9TP [PNAME=]*pdn* [keywords] 'external id'

The form of the request for mounting a MERIT network connection is:

MNET [PNAME=]*pdn* [keywords]

Description: For mounting magnetic tapes, the user must specify a rack number and an external identification. The rack number is the location number of the tape reel in the Computing Center tape library. The 'external id' is the name associated with the external label. This label is typed on a gummed sticker and is attached to the reel when it is initially assigned to a user or submitted to the Computing Center. If the information between the single quotes cannot be judged by the computer operator as reasonably corresponding to the external identification on the tape, the operator informs the user that the tape is unavailable. If a scratch tape is to be mounted, the word POOL must be used in place of the rack number and no external identification is required.

The user must specify a device type which indicates on which type of device the item is to be mounted. The device types available are:

9TP - 9-track magnetic tape
MNET - MERIT network connection

The user must specify a psuedo-device name "pdn" for each item to be mounted. After the item is mounted, all command and program references to the item are made using its "pdn",

rather than the name of the actual device on which the item is mounted. The "pdn" is used in the same context as a file or device name (FDname). A "pdn" begins with an asterisk, ends with an asterisk, has from one to fourteen characters in between, and must not conflict with pre-defined MTS "pdns" such as *SOURCE*, *SINK*, and so on. (See the section on pseudo-device names in "Files and Devices" in this volume.)

The user may specify many different keyword parameters as part of the mount request. Successive keyword parameters may be entered in any order and must be separated by blanks or commas. For a complete description of the effect of each of the keyword parameters, see the "Magnetic Tape User's Guide" in Volume 4. The tables on the following pages summarize the keyword parameters currently available. The default values are underlined where appropriate.

Items mounted by the $MOUNT command may be dismounted by the $RELEASE command.

Examples:

```
$MOUNT
1234 9TP *TAPE* VOL=001234 RING=IN 'TEST TAPE #1'
POOL 9TP *POOL*
$ENDFILE
```

The labeled magnetic tape 1234 is mounted on a 9-track magnetic tape unit; the volume label of the tape is 001234 and the external identification is TAPE TEST #1; the tape is mounted with the file protect ring in; the tape is assigned a pseudo-device name *TAPE*. A pool tape is also mounted on a 9-track tape unit; it is assigned the pseudo-device name *POOL*.

```
$MOUNT MNET *W* DEST=MS
```

A MERIT network connection to Michigan State University is mounted and assigned the pseudo-device name *W*.

Keyword Parameter Summary

For magnetic tapes:

{BLOCKING|BLK}={ON|OFF}	Enables or disables blocking
{DSNAME|DSN}=name	Data set name to be used for first new file written
{FORMAT|FMT|RECFM}=fmt[([size][,lrecl])]	Specifies blocking format and, optionally, block size and/or logical record length
LP={ON|OFF}	Enables or disables label processing
LRECL=n	Specifies logical record length ($1 \leq n \leq 32767$)
MODE={800|1600}	Specifies tape mode
NEWEXP={yymmdd|NONE}	Initializes, changes or deletes the expiration date for writing
NEWRPW={psswrd|NONE}	Initializes, changes or deletes the read password
NEWWPW={psswrd|NONE}	Initializes, changes or deletes the write password
OVRIDE=OK	Overrides existing expiration date for writing
POSN={*n*|*EOT*|name}	Positions tape to the nth file, end-of-tape or data set name
QUIT={YES|NO}	Controls termination of batch job if mount fails
RERUN={YES|NO}	Specifies rerunning of a batch job if no drives are available
RETRY=n	Specifies retry count for read errors ($0 \leq n \leq 15$)
RING={IN|OUT}	Specifies placement of file protect ring
RPW=psswrd	Protects reading tapes with RING=OUT

{SIZE|BLKSIZE}=n	Specifies maximum block size (18≤ n ≤ 32767)
{VOLUME|VOL|LABEL|VOLSER}=name	Volume name of labeled tape
WPW=psswrd	Protects writing tapes with RING=IN
WRITE={YES|NO}	Specifies placement of file protect ring

For MERIT network connections:

{DEST|D}={MS|UM|MS}	Specify connection destination
QUIT={YES|NO}	Control termination of batch job if mount fails.

April 1974

NET

COMMAND DESCRIPTION

Purpose: To enter network mode.

Prototype: $NET [*pdn*] [network-command...]

pdn

"*pdn*" is the pseudo-device name of a previously acquired MERIT network connection. If the pseudo-device name is omitted or invalid, the user is prompted for it. If the user previously left network command mode via the .MTS network command, the previously used connection is maintained unless a new pseudo-device name is specified. Network connections are acquired via the $MOUNT command.

network-command

"network-command" is an optional network command line to be passed to network command mode upon entry.

Description: See "The MERIT Network User's Guide" in Volume 4 for a complete description of how to use network mode with the MERIT computer network.

Examples: $NET *W*

Network mode is entered for the pseudo-device *W*.

$NET

Network mode is entered for the last pseudo-device used on a $NET command.

NUMBER

COMMAND DESCRIPTION

Purpose: To start automatic numbering of input lines being read from *SOURCE* and being written on the currently active file (*AFD*).

Prototype: $NUMBER [par]

One of the two following parameter sequences may be given:

[starting-number] [[,]increment]

The starting line number for automatic line numbering is given by "starting number". If this is omitted, line numbering begins with 1. The starting line number can take any one of the three forms:

line number
LAST
LAST±number

where LAST is the line number of the last line in the currently active file. If the file is empty, the value of LAST is zero.

The line number increment is given by "increment". It may be specified in the same manner as the "starting number". If "increment" is omitted, line numbers are incremented by 1. The parameter "increment" must be separated from "starting number" by a blank or a comma. If only "increment" is given, it must be preceded by a comma.

CONTINUE

The parameter CONTINUE resumes automatic numbering from the point where automatic numbering was when last terminated by the $UNNUMBER command. The release or change of the currently active file affects automatic numbering by causing the numbering to start at line number 1 when CONTINUE is specified. If CONTINUE is specified and automatic numbering has not been previously turned on, the starting line number and the increment defaults to 1.

Description: Provided there is a currently active file, any input line not recognized as a command line has a line number automatically assigned to it before it is written on the currently active

file. In the absence of automatic numbering, this line number is taken from the first characters of the input line if they are valid as a line number. If they are not valid, the line is treated as an invalid command. The NUMBER command turns on automatic numbering of the input lines being read from *SOURCE*. In terminal mode, this number is printed at the front of the input line in the form:

```
#number
#     1_
#     2_
```

(This printing of the "#" may be turned off by the $SET PFX=OFF command.) The line numbers provided by the automatic numbering sequence are taken as the line number for the input line; the first characters of the line are treated as an integral part of the line. The line number assigned to the input line is used for an indexed write operation to the currently active file. Therefore, the currently active file should be a line file or a device. If the currently active file is a sequential file, the command $SET SEQFCHK=OFF must be issued first; however, the line number assigned to the input line is ignored and the line is written sequentially at the end of the file.

In terminal mode, when automatic numbering is turned on and there is an active file, all command lines must begin with a $. Consequently the UNNUMBER command must always be preceded by a $.

Examples:

```
$NUMBER
```

This turns on automatic numbering with a starting line number of 1 and an increment of 1.

```
$NUM 10,5
```

This turns on automatic numbering with a starting line number of 10 and an increment of 5.

```
$NUM -100
```

This turns on automatic numbering with a starting line number of -100 and an increment of 1.

PERMIT

COMMAND DESCRIPTION

Purpose: To allow users to access other user's files.

Prototype: $PERMIT [filename] [access type]

filename

"filename" is the name of the file to be permitted. If "filename" is omitted, the user is prompted to enter a filename and access-type. Filenames and access-types should be listed, one set per line, terminated by a null line or an end-of-file.

access-type

"access type"" is one (or more) of the following:

ALL	All other users are allowed to read the file.
RUN	The file may only be run or loaded. It may not be copied or listed.
PROJNO	The file may be read (using *COPY) by all other users having the same project number as the owner.
NONE	The owner has unlimited access. All other users have no access.
CNW	The file may not be written or emptied.
RO	RO is equivalent to CNW and ALL.

If no access type is specified, NONE is assumed.

Description: The above information is for reference only. For a general description of the various parameters, see the "Shared Files" section of the "Files and Devices" description in this volume.

Examples: $PERMIT X ALL

All users are given read access to file X.

$PERMIT Y CNW RUN PRJNO

All users with the same project number as the owner are allowed run only access to file Y. The owner can no

longer change the file (although he can still destroy it).

$PERMIT X

All permit access is removed from file X.

RELEASE

COMMAND DESCRIPTION

Purpose: To release the currently active file or device.

Prototype: $RELEASE [*pdn*]

pdn

"*pdn*" is either *AFD*, the pseudo-device name from a mount request to $MOUNT, *PRINT*, *PUNCH* , or *BATCH*.

Description: If the parameter is omitted or is *AFD*, the currently active file or device is closed and the pseudo-device *AFD* is disassociated from the file or device and becomes undefined.

The RELEASE command is used in this case after all changes to a file have been made. It protects the user from accidentally entering data into the file through the misspelling of a command, or in batch mode, through having a program terminate abnormally and leaving as yet unread data cards which could be mistakenly entered into the active file.

If the parameter specifies a pseudo-device name for a mounted device, that device is dismounted.

If the parameter specifies *PRINT*, *PUNCH*, or *BATCH*, the associated job is released to HASP. This is the same as if the command

$CONTROL *...* RELEASE

was given.

Example: $REL

RESTART

COMMAND DESCRIPTION

Purpose: To restart (or initiate) execution of a program following either initial loading, an interrupt, or a subroutine call to ERROR, MTS or MTSCMD.

Prototype: $RESTART [[AT] location] [MAP[=mapFDname]] [NOMAP] [XREF] [I/OFDnames] [limits]

The following parameters may be given:

[AT] location

The address at which execution is to begin is specified by "location". "location" is a virtual memory location given by an optional local relocation factor and a displacement in the form

[RF={hhhhhh|GRx}] xxxxxx

where "hhhhhh" is the hexadecimal value of a local relocation factor or GRx indicates the general register whose contents are to be used as a local relocation factor, and "xxxxxx" is the hexadecimal value of a displacement. The displacement is added to the current value of the relocation factor to provide an absolute virtual memory address. If a local relocation factor is not specified, the global relocation factor is used. The global relocation factor is initially zero, but may be changed by the $SET command. Since this value replaces the right-hand 32 bits of the PSW, "location" specifies the instruction length code, the condition code, and the program mask as well as the displacement.

MAP[=mapFDname] [NOMAP] [XREF]

The user can redesignate the destination of future load maps with the MAP keyword. This is only useful if the program is loading other modules dynamically by calling the subroutines LOAD, LINK, or XCTL. The NOMAP keyword can be used to suppress the printing of all future load maps. See the RUN command description.

I/OFDnames

The user can reassign the logical I/O units. See the RUN command description.

FORTRAN users are reminded that MTS logical I/O units 0 through 19 are not necessarily the same as the FORTRAN logical I/O units 0 through 19. The FORTRAN I/O routines may default the FORTRAN logical I/O units independently of MTS. Moreover, if this has happened, the meaning of these units cannot be reassigned by a RESTART command.

limits

The user can specify local limits for CPU time, pages printed, and cards punched. See the RUN command description. If no limits are specified, no limits are enforced. Any limits specified on previous RUN or RESTART commands are ignored.

Description: The RESTART command restarts (or initiates) execution of the currently loaded program. This may be a program loaded by the LOAD command, a program that was interrupted during execution by a program or attention interrupt, or a program that terminated by a subroutine call to ERROR, MTS or MTSCMD.

A 32-bit value is computed from the "location" specification and replaces the right-hand 32 bits of the PSW. If "location" is omitted, the PSW remains unaltered, and execution begins at the entry point for a program loaded by the LOAD command, or the point of interruption for a program that is interrupted by a program or attention interrupt; for a program that is interrupted by a call to ERROR, MTS or MTSCMD, execution restarts at the point following the subroutine call as if the subroutine returned to the program.

If logical I/O units have been reassigned, the files and devices originally assigned are closed, and the newly assigned files and devices are opened.

Examples: `$RESTART SPRINT=A`

This restarts the currently loaded program with SPRINT output reassigned to the file A.

`$RE AT RF=520800 28000258`

The program is restarted at location 520A58 with the condition code set to 2, fixed point overflow interruption enabled, and the other program mask interrupts disabled.

RUN

COMMAND DESCRIPTION

Purpose: To load and initiate execution of a program.

Prototype: $RUN [objectFDname] [MAP[=mapFDname]] [NOMAP] [XREF]
[I/OFDnames] [limits] [PAR=parameters]

The following parameters may be given:

objectFDname

"objectFDname" specifies the file(s) or device(s) containing the program(s) to be loaded. If omitted, the program is loaded from *SOURCE*.

MAP[=mapFDname] [NOMAP] [XREF]

The MAP keyword specifies the file or device "mapFDname" on which the loader is to write the load map. If the MAP keyword parameter is omitted, no load map is written. If the MAP keyword is given in the form of MAP, "mapFDname" defaults to *SINK*. The NOMAP parameter suppresses the printing of the load map. NOMAP need be given only if a previous MAP specification was made.

The XREF parameter specifies that a cross reference listing of external symbols occurring in the loaded programs is to be produced in addition to the load map. If XREF is given without the MAP keyword, MAP=*SINK* is assumed. If XREF is given with NOMAP, only the cross reference is produced.

I/OFDnames

The keyword parameters "I/OFDnames" are the assignments of logical I/O units to files or devices for use by the loaded program during execution. The logical I/O unit assignments establish the I/O subroutines which are used by the loaded program for the input and output of data. Where no specifications are stated, the following default assignments occur:

```
SCARDS=*SOURCE*
SPRINT=*SINK*
SPUNCH=*PUNCH* (batch mode if global card est. >0)
SERCOM=*MSINK*
GUSER=*MSOURCE*
```

The logical I/O units 0 through 19 have no default specifications. See the section "Files and Devices" in this volume and the subroutine descriptions for SCARDS, SPRINT, SPUNCH, SERCOM, GUSER, READ and WRITE in Volume 3 for further details on the use of these subroutines.

FORTRAN users are reminded that MTS logical I/O units 0 through 19 are not necessarily the same as the FORTRAN logical I/O units 0 through 19. The FORTRAN I/O routines may default the FORTRAN logical I/O units independently of MTS. Moreover, if this has happened, the meaning of these units cannot be reassigned by a RESTART command.

limits

The keyword parameters "limits" specify local limits for CPU time, pages printed and cards punched. These can be given in the form:

TIME={t|tS|tM}
PAGES=p
CARDS=c

These local limits can be used in both batch and terminal mode and are effective only for the execution of the program loaded by the $RUN command. See the description of the similar global limits in the $SIGNON command description for further details of limit specifications.

PAR=parameters

The PAR keyword specifies an arbitrary string of characters to be passed to the loaded program on initiation of execution. This is usually a parameter list for the program and its interpretation depends on the loaded program. The PAR keyword must be the last parameter field specified in the command. The parameter list is terminated by the end of the command line.

Description: The RUN command calls upon the dynamic loader to load the object program into virtual memory. If there are unresolved external symbol references after loading from "objectFDname", loading continues from *LIBRARY (the system library). Only those parts of *LIBRARY required to resolve the references are loaded. If there are still unresolved external references, a fatal loading error exists. In terminal mode, the loader prompts for more loader input; in batch mode, an error comment is produced and the loading terminates immediately. The search of the system library may be suppressed by the $SET LIBR=OFF command. If $SET DEBUG=ON has been

specified, SDS processes any symbol table information associated with the loaded program.

If there were no fatal loading errors, the comment "EXECUTION BEGINS" is printed and control is transferred to the entry point of the program by a standard subroutine call (the entry point address in GR15, the return address in GR14, the save area location in GR13 and the parameter list location in GP1).

The parameter string (specified by the PAR keyword) is passed as follows: GR1 contains the location of a fullword address constant which points to a region containing a halfword count (halfword aligned) followed by an EBCDIC character region (of byte-length specified by the count) containing the parameter string. The leftmost bit of the address constant is 1.

If files or devices specifed by "I/OFDnames" are non-existent or are not available, an error comment is produced and the logical I/O unit referring to the unavailable file or device is set up such that the first time the program being executed refers to that logical I/O unit, either the user is given an opportunity to change the FDname (in terminal mode) or execution is terminated (in batch mode).

If the program terminates execution by restoring the registers and returning to MTS via GR14 or by calling the SYSTEM subroutine, the comment "EXECUTION TERMINATED" is printed, unless $SET DEBUG=ON has been given.

All storage, files, and devices used for this RUN command are automatically released unless the user has issued the $SET UNLOAD=OFF command or execution has not terminated normally (for example, the program calls the ERROR subroutine or an attention or program interrupt has occurred).

If storage, files, and devices are not released, the user can use the commands $DISPLAY, $ALTER, and $RESTART to debug or continue the program.

Examples:

```
$RUN -LOAD
```

This loads and initiates execution of the program in the temporary file -LOAD which could, for example, contain the object module from a FORTRAN compilation using *FTN.

```
$R OBJPROG+*SSP MAP 5=INPUT 6=OUTPUT
```

This loads and initiates execution of the program in OBJPROG which presumably contains references to subroutines in *SSP. A load map is printed on *SINK*. Logical I/O units 5 and 6 are specified for the input and output of data to the program.

```
$RUN *WATFOR T=10
```

This loads and initiates execution of the Waterloo FORTRAN program. A local time estimate of 10 seconds is specified.

```
$R *ASMG SCARDS=SOURCE SPUNCH=OBJPROG 0=*SYSMAC 2=MACROS -
     PAR=IBLK=40,T,FX,RD
```

This loads the 360 assembler *ASMG and initiates execution of the assembler with source input from the file SOURCE and object module output to the file OBJPROG. The macro libraries *SYSMAC and MACROS are attached to logical I/O units 0 and 2. The parameter string in the PAR field is passed to the assembler by the RUN command.

April 1974

SDS

COMMAND DESCRIPTION

Purpose: To enter or return to debug mode.

Prototype: $SDS [debug-command]

Description: Control is transferred to debug command mode. In debug mode, the user has the facilities of SDS at his disposal. See the section "Debug Mode" for a description of the symbolic debugging system.

If a debug command is specified with the SDS command, that debug command is executed, and control is returned immediately to MTS command mode.

Examples:

```
$SDS
```

Control is transferred to debug mode.

```
$SDS SET ERRORDUMP=ON
```

The automatic errordumping option of SDS is enabled; control is returned to MTS command mode.

SET

COMMAND DESCRIPTION

Purpose: To set various global switches and parameters.

Prototype: $SET keyword ...

Any number of the following keyword parameters may be given in a single $SET command. The keyword parameters should be separated by blanks; there must be no blanks within any keyword parameter.

All of the values set by these keywords may be interrogated by programs using the GUINFO subroutine and may be changed by programs using the CUINFO subroutine. See the subroutine description for GUINFO and CUINFO in Volume 3 for details.

AFDECHO={ON|OFF} Default: OFF

If the AFDECHO keyword is ON, all lines written to the active file in command mode are echoed to *SINK* and *MSINK*. Each line echoed is preceded by its line number in the active file.

CASE={UC|LC|MX|MIXED} Default: LC

The CASE keyword specifies global uppercase (UC) or lowercase (LC) conversion. If UC is specified, all data lines read by the MTS monitor have lowercase letters converted to uppercase letters. If LC is specified, the data lines are not converted. MX and MIXED are equivalent to LC.

CMDSKP={ON|OFF} Default: OFF

Normally, in printed output from a batch run, a page-skip is produced automatically following any commands that may generate printed output (LIST, COPY, RUN, and so on). If the CMDSKP keyword is OFF, only a triple-space is produced, if CMDSKP is ON, the page-skip is produced.

CONTCHAR=character Default: -

The CONTCHAR keyword specifies a single character to be used as an indicator for continuing an MTS command line on another input line. The continuation character must be the last character of the line to be continued.

COST={ON|OFF} Default: OFF

If the COST keyword is ON, the cost since the last cost was printed (not necessarily the cost of the last command) and the total current cost of the session is printed; these costs then continue to be printed after every command. These costs are approximate and subject to roundoff; items which are not immediately charged for such as tape drive time are not included until the actual charge is made. If cost is off, these costs are not printed. This keyword is the same as the $ keyword described below.

CREAFD={ON|OFF} Default: ON

If the CREAFD keyword is ON, a successful $CREATE command causes the created file to become the currently active file. If CREAFD is OFF, the created file is not made the currently active file.

DEBUG={ON|OFF} Default: OFF

If DEBUG is ON, all programs loaded are processed by SDS for any symbol table information associated with them. This allows debug mode commands to refer to symbols in programs compiled with the TEST option. Also, all programs run in execution mode are monitored by SDS and any errors produce debug mode error comments.

DEVCHAR=character Default: >

The DEVCHAR keyword specifies a single character to be used to indicate that a following FDname is a device, not a file.

ECHO={ON|OFF} Default: ON

If the ECHO keyword is ON, MTS command lines are echoed on the *SINK* or *MSINK* devices (or files) if these devices (or files) are different from the *SOURCE* device (or file). If ECHO is OFF, echoing does not occur.

EDITAFD={ON|OFF} Default: OFF

If the EDITAFD keyword is ON, the $EDIT command causes the file to be edited to become the currently active file. If EDITAFD is OFF, the file to be edited is not made the currently active file. This keyword may also have other effects on the $EDIT command. See the $EDIT command in this volume for details.

ENDFILE={ON|OFF|NEVER} Default: OFF

If the ENDFILE keyword is ON, a $ENDFILE line is recognized as an end-of-file whenever it is read; if ENDFILE is OFF, a $ENDFILE line is recognized as an end-of-file only when read from *SOURCE* or *MSOURCE*; if ENDFILE is NEVER, a $ENDFILE is never recognized as end-of-file.

ERRORDUMP={ON|OFF|FULL} Default: OFF

If the ERRORDUMP keyword is ON and execution terminates abnormally in batch mode, a storage dump of the user's loaded program is given; if ERRORDUMP is FULL, the storage dump includes any library subroutines loaded; if ERRORDUMP is OFF, no dump is given. ON is the same as FULL unless "$SET SYMTAB=ON" was specified before the program was loaded. The ERRORDUMP keyword has no effect in terminal mode.

FILECHAR=character Default: #

The FILECHAR keyword specifies a single character to be used to indicate that a following FDname is a file, not a device.

IC={ON|OFF} Default: ON

If the IC keyword is ON, the line "$CONTINUE WITH" specifies implicit concatenation; if IC is OFF, this line is treated as a data line. The IC keyword can be overridden by the @IC modifier on I/O operations. (See the appendix "I/O Modifiers" to the section "Files and Devices" in this volume.)

LIBR={ON|OFF} Default: ON

If the LIBR keyword is ON, the file *LIBRARY and LCSYMBOL (the low-core symbol directory) are searched for unresolved external symbols after a program is loaded; if LIBR is OFF, this search is not made.

LIBSRCH={OFF|FDname} Default: OFF

The LIBSRCH keyword indicates what (if any) public or private libraries are to be searched if there are unresolved symbols in a loaded program. This keyword interacts with the LIBR keyword. If LIBR is OFF, then the LIBSRCH keyword is ignored. If LIBR is ON, but LIBSRCH is OFF, then only *LIBRARY (the system library) and LCSYMBOL (the low-core symbol directory) is searched (if the *LIBRARY keyword is ON). If LIBR is ON and LIBSRCH is not OFF, then LIBSRCH specifies a library or

concatenation of libraries to be searched. The libraries are searched in the order specified in the keyword. After these libraries are searched, *LIBRARY and LCSYMBOL are searched if there are still any unresolved symbols (if the *LIBRARY keyword is ON).

LNS=character Default: ,

The LNS keyword specifies a single character to be used as a line number separator. The line number separator is used to terminate the line number for the input line read in MTS command mode and indicates that the remaining characters are a part of the data line itself. This is used to distinguish a line beginning with numeric information from the line number (for example, 174, 112233445566 has a line number of 174 and a data line of 112233445566).

PFX={ON|OFF} Default: ON

If the PFX keyword is ON, the terminal prefix character is printed at the front of each input or output line at the user's terminal, or if automatic numbering is on (due to the NUMBER command) and there is an active file, the line number that is assigned is printed at the front of each input line at the user's terminal; if PFX is OFF, the printing of the prefix character or line number is suppressed. The terminal prefix character is used to determine which programs are communicating with the user. The prefix character is not printed in batch mode.

PW=characters

The PW keyword specifies a sequence of one to six characters (none of which are blank) to be used as the user's password. The user's password must be correctly given before the user is allowed to sign on.

RF={hhhhhh|GRx} Default: RF=000000

The RF keyword sets a global relocation factor which is used in the $DISPLAY and $RESTART commands. "hhhhhh" is a hexadecimal constant to be used as the global relocation factor or GRx specifies a register whose contents are to be used as the relocation factor. The relocation factor is set to zero initially. If GRx is specified, the current contents of that register at the time of the SET command is used as the relocation factor.

SCRFCHAR=character Default: -

The SCRFCHAR keyword specifies a single character to be used to indicate that a following FDname is a temporary file, not a permanent file.

SEQFCHK={ON|OFF} Default: ON

Normally an attempt to do an indexed operation on a sequential file or an attempt to do a sequential operation starting at other than line 1 on a sequential file causes an error condition and an error message to be generated. If SEQFCHK is OFF, the message is not issued and the operation is performed as if not indexed.

SHFSEP=character Default: :

The SHFSEP keyword specifies a single character to separate the signon ID from the file name when referring to a shared file (for example, 1AGA:DATAFILE).

SIGFILE={OFF|FDname} Default: OFF

The SIGFILE keyword indicates which file (if any) is the special SIGNON file used as an implicit $SOURCE file after the user signs on. If SIGFILE is set to OFF, the file or device still exists, but is not used as a SIGNON file. The setting of SIGFILE affects subsequent signons.

SIGFILEATTN={ON|OFF} Default: ON

If the SIGFILEATTN keyword is ON, an attention interrupt during the "last sign-on" message or while the SIGNON file is being processed, interrupts the processing and resets SOURCE to MSOURCE. If SIGFILEATTN is OFF, attention interrupts are stacked, but are not taken during the SIGNON file processing (except if the SIGNON file runs a program which calls the subroutine ATTNTRP, the attention is taken at that point and given to the program). The setting of SIGFILEATTN affects subsequent signons.

SYMTAB={ON|OFF} Default: ON

If the SYMTAB keyword is ON, the loader symbol table is retained whenever a program has been loaded. This allows external symbols used in a program to be used by MTS and other user programs. If SYMTAB is OFF, the loader symbol table is not retained.

TDR={ON|OFF} Default: OFF

If the TDR keyword is ON, the CPU time (in seconds) and the number of drum reads which have occurred since the last MTS command was terminated, is printed on *SINK*. This allows a user to easily obtain a timing estimate for the execution of a program.

TERSE={ON|OFF} Default: OFF

If the TERSE keyword is ON, many informational and error messages from MTS are suppressed or abbreviated. If TERSE is OFF, all messages are given in full.

TRIM={ON|OFF} Default: ON

If the TRIM keyword is ON, lines read or written to files are trimmed (trailing blanks are removed). If TRIM is OFF, the lines are not trimmed. The TRIM keyword can be overridden by the @TRIM modifier on I/O operations. (See the appendix "I/O Modifiers" to the section "Files and Devices" in this volume.)

UNLOAD={ON|OFF} Default: ON

If the UNLOAD keyword is ON, all storage, files and devices from a previous LOAD or RUN command are automatically released at the end of normal execution; if UNLOAD is OFF, they must be released using the UNLOAD command.

*LIBRARY={ON|OFF} Default: ON

If the *LIBRARY keyword is ON, *LIBRARY is searched for any unresolved external symbols after a program is loaded (if the LIBR keyword is ON). If *LIBRARY is OFF (or if LIBR is OFF), this search is not made.

$={ON|OFF} Default: OFF

If the $ keyword is ON, the cost since the last cost was printed (not necessarily the cost of the last command) and the total current cost of the session is printed; these costs then continue to be printed after every command. These costs are approximate and subject to roundoff; items which are not immediately charged for such as tape drive time are not included until the actual charge is made. If $ is OFF, these costs are not printed. This keyword is the same as the COST keyword described above.

Examples:

```
$SET IC=OFF ENDFILE=NEVER
```

This forces the lines "$CONTINUE WITH" and "$ENDFILE" to be interpreted as data lines, not as implicit concatenation indicators or as end-of-file.

```
$SET ECHO=OFF
$SET PW=AJAX ECHO=ON
```

This batch example sets the user's password to AJAX. The echoing of MTS command lines is turned OFF so that the password is not echoed on the user's output.

```
$CREATE <SIGFILE>
$GET <SIGFILE>
1$$SET COST=ON ERRORDUMP=ON
$SET SIGFILE=<SIGFILE>
```

This example creates a file <SIGFILE> to be used as a SIGNON file. This SIGNON file contains a SET command to set the COST and ERRORDUMP options to ON. Each time the user signs on, this file is invoked causing the specified options to be set.

SIGNOFF

COMMAND DESCRIPTION

Purpose: To notify the system of a user's departure.

Prototype: $SIGNOFF [SHORT|$]

The only legal parameters are: SHORT or $

An abbreviated form of the sign off statistics is produced when the SHORT keyword is given. If $ is specified, only the time and date of the user's signoff, the approximate cost of the session, and the user's remaining balance of funds is printed.

Description: The user is signed off the system. All storage acquired and devices attached are released, and all files are closed (temporary files are destroyed). A summary of the job statistics is printed. These statistics include all of the non-zero entries from the following items:

User SIGNON ID (batch only)
Project number (batch only)
Time of sign-on (batch only)
Time of sign-off
Elapsed time in minutes
CPU time used in seconds
CPU storage virtual memory integral in page-minutes
Wait-state virtual memory integral in page-hours
Number of cards read
Number of lines printed
Number of pages printed
Number of cards punched
Number of tape mounts
Tape drive time used in minutes
MERIT network time
Number of drum reads
Approximate total cost of the job
Charge for plotting time
Charge for disk space used since last sign-off
Remaining balance of funds for the ID
Time of the previous sign-on (batch only)

If the abbreviation $SIG is used, its meaning is taken in context: if the user is not signed on, $SIG means $SIGNON; if the user is signed on, $SIG means $SIGNOFF.

Examples:

```
$SIGNOFF
$SIG $
$SIG S
```

April 1974

SIGNON

COMMAND DESCRIPTION

Purpose: To identify a user to the system.

Prototype: $SIGNON ccid [keywords] ['d name']

Only the "ccid" parameter giving the user's signon identification number is required. The other legal keyword parameters that may be given are:

PW=password

The PW keyword specifies the user's password. This keyword may be used in both batch and terminal mode; however, it is recommended that batch users do not use this keyword, but supply the password (without PW=) on the card following the $SIGNON command. In this way, the password does not appear on the user's printed output.

The following keywords are relevant in batch mode only. They are useful for placing global constraints on the user's job.

TIME={t|tS|tM} Default: 30S

The TIME keyword gives a number "t" specifying the global CPU time limit for the job. If given in the form of the number "t" or "tS", the time limit is in seconds; if given in the form "tM", the time limit is in minutes. If the TIME keyword is not specified, a default time limit of 30 seconds is enforced.

PAGES=p Default: 50

The PAGES keyword gives a number "p" specifying the global printed page limit for a single copy of the job. If the PAGES keyword is not specified, a default page limit of 50 pages is enforced. The maximum value that can be specified is 99999 pages.

CARDS=c Default: 0

The CARDS keyword gives a number "c" specifying the global punched card limit for the job. If the CARDS keyword is not specified, a default card limit of 0 is enforced. The maximum value that can be specified is 99999 cards.

COPIES=n Default: 1

The COPIES keyword gives a number "n" specifying the number of printed copies of the output to be produced. If the COPIES keyword is not specified, one copy of the output is printed.

PRINT={QN|TN} Default: any standard train

The PRINT keyword specifies the type of character set to be used for printing the job. This may be either the QN or TN character set. The normal character set is QN. The TN character set contains a larger set of characters (including lower-case alphabetics) and is correspondingly slower in printing output. For a description of the character sets, see the "Print Character Sets" section in Volume 5 of the MTS manuals.

ROUTE=locn
PROUTE=locn
CROUTE=locn Default: location where job was submitted.

These three keywords specify the destination for printed and punched output. "locn" is a four character code specifying the location at which the job's output is to be printed and/or punched. The ROUTE keyword sends all output to "locn", while the PROUTE and CROUTE keywords send the printed and punched output, respectively. Currently, CNTR is the code for the Computing and Data Processing Center and SCIL is the code for the Science Library basement. If any station specified does not have a card punch, the punched output is re-routed to CNTR. The default for all *BATCH*, *PRINT*, and *PUNCH* jobs is CNTR.

'd name'

This comment field should contain a delivery code "d" (see Appendix B of the Facilities and Services of WSU Computing and Data Processing Center for a list of valid delivery codes), and user identification "name" which may actually be the user's name, course and section number, etc. If the output is to be retained at the CDPC Control Desk rather than delivered, the first character of the comment field must be a blank. The comment must be the last item of the $SIGNON command and must be enclosed in single quote marks.

Description: The $SIGNON command identifies the user to the system (that is, signs him on), and in the case of batch jobs, establishes certain constraints for the job. The $SIGNON command must be the first command of the user's job.

If the user has not specified a password on the $SIGNON command, the system prompts him for the password at a terminal. For a batch job, the system expects to find the password starting in column 1 on the next card following the $SIGNON card.

A user may not be signed on more than once at any one time for any given "ccid".

If the abbreviation $SIG is used, its meaning is taken in context: if the user is not signed on, $SIG means $SIGNON; if the user is signed on, $SIG means $SIGNOFF.

Examples:

```
$SIGNON 2AGA '  JOHN Q. DOE'
```

The user with ID 2AGA is signed on to the system. All global limits and output specifications have default values.

```
$SIG 2AGA T=1M P=100 C=50 'P JOHN Q.  DOE'
```

The user with ID 2AGA is signed on to the system. The global limits are 1 minute of CPU time, 100 pages of printed output, and 50 punched cards. The deck and output is delivered to the Physics Department distribution center.

```
$SIG 2AGA T=20 COPIES=2 PRINT=TN '  JOHN Q. DOE'
```

The global time limit specifed is 20 seconds. Two copies of the printed output are produced using the TN character set.

```
$SIG 2AGA C=1000 PROUTE=SCIL 'W JOHN Q. DOE'
```

A batch job submitted elsewhere is printed at SCIL (Science Library). The punched output is punched at CNTR and is delivered to location W (Science Library) along with the input deck.

SINK

COMMAND DESCRIPTION

Purpose: To change the destination or "sink" for normal output lines.

Prototype: $SINK {FDname|PREVIOUS}

One of the two following parameters must be given:

FDname

The name of the file or device that is to become the current sink of output lines.

PREVIOUS

The PREVIOUS parameter specifies that the previous sink is to be restored as the current sink. To output to a file by the name PREVIOUS, the file character # must precede the file name.

Description: When the $SINK command is given, the pseudo-device *SINK* is reassigned to the file or device specified. This causes output which defaults to *SINK* to be sent to that file or device. The master sink *MSINK* remains as the terminal in terminal mode or the printer in batch mode. Initially *SINK* is assigned to the same device as *MSINK*.

Error messages requiring user interaction are directed to *MSINK* (terminal mode only).

An attention interrupt causes *SINK* to revert to *MSINK*.

A one level pushdown list of sink devices is maintained. The PREVIOUS parameter uses this pushdown list to restore the previous sink device.

Examples: $SINK A

The file A becomes the current sink for output lines.

$SINK PREVIOUS

The previous file or device used as the sink becomes the current sink for output lines.

SOURCE

COMMAND DESCRIPTION

Purpose: To change the source of input lines.

Prototype: $SOURCE {FDname|PREVIOUS}

One of the two following parameters must be given:

FDname

The name of the file or device that is to become the current source of input lines.

PREVIOUS

The PREVIOUS parameter specifies that the previous source is to be restored as the current source. To source to a file by the name PREVIOUS, the file character # must precede the file name.

Description: When the $SOURCE command is given, the pseudo-device *SOURCE* is reassigned to the file or device specified. This causes the next input line to be taken from that file or device. Input that defaults to *SOURCE* is read from that file or device. The master source *MSOURCE* remains as the terminal in conversational mode or the card reader in batch mode. Initially *SOURCE* is assigned to the same device as *MSOURCE*.

Responses to error messages requiring user interaction are read from *MSOURCE*.

An attention interrupt, or an end-of-file condition on *SOURCE* when trying to read an MTS command, causes *SOURCE* to revert to *MSOURCE*.

A one level pushdown list of source devices is maintained. The PREVIOUS parameter uses this pushdown list to restore the previous source device.

Example: $SOURCE A

The file A becomes the current source for input lines.

April 1974

START

COMMAND DESCRIPTION

Purpose: To restart (or initiate) execution of a program following either initial loading, an interrupt, or a subroutine call to ERROR, MTS, or MTSCMD.

Prototype: $START [[AT] LOCN] [MAP[=mapFDname]] [NOMAP [XREF] [I/OFDnames] [limits]

This command is identical to the $RESTART command.

UNLOAD

COMMAND DESCRIPTION

Purpose: To unload the currently loaded program in virtual memory.

Prototype: $UNLOAD [CLS=xxx]

Description: The $UNLOAD command unloads the current program which had previously been loaded by a $LOAD command or a $RUN command if execution did not terminate normally. (Normal termination is by the program returning to the system with a return code of zero or a subroutine call to SYSTEM.) All storage allocated to the program is released, and all files and devices opened at execution time are closed.

The parameter "CLS=xxx" releases all space associated with the command language subsystem "xxx". This should be used only if the CLS is not working properly.

Example: $UNLOAD

April 1974

UNNUMBER

COMMAND DESCRIPTION

Purpose: To turn off automatic numbering of input lines from *SOURCE*.

Prototype: $UNNUMBER

Description: Automatic number of input lines being read from *SOURCE* is turned off. Any input line, not recognized as a command line or a data line (starts with a valid line number), is treated as an invalid command.

Example:

```
#number
#      1_this
#      2_is data
#      3_$unnumber
```

Automatic line numbering was started with beginning line number of 1 and an increment of 1. Three lines later automatic numbering was suspended with the $UNNUMBER command. Automatic numbering may be resumed with a starting line number of 3 and an increment of 1 by issuing the NUMBER CONTINUE command.

April 1974

EDIT MODE

BASIC CONCEPTS

The MTS context editor program is used for the editing of MTS line files. The editor is invoked by the command

$EDIT filename

where "filename" is the name of the MTS line file to be edited. (See the MTS $EDIT command description in this volume for details.) Commands to the editor are read from the pseudo-device *SOURCE* and editor output messages, diagnostics, and verification comments are written on *SINK*. After the file "filename" has been obtained, the editor responds with a colon ":" at which point it is ready to accept edit commands.

Files

References to line numbers within a file by the edit commands and references to line numbers within a file by MTS file naming conventions are the same. Hence limits of line numbers and file sizes applicable to MTS conventions apply to the editor as well. The specification of line number ranges and explicit concatenation for the file are ignored by the editor. The file is read with the I/O modifiers implicit concatenation off and trimming off (¬IC and ¬TRIM). The file is written with trimming off.

The editor program works only with MTS line files. Text editing of sequential files and sequential files with line numbers is not supported by this program.

Column Ranges

Various edit commands which perform some action on lines of the files being edited act only within a specified "column range", an interval of each line. Two column pointers are associated with the column range which is defined as extending from the character pointed to by the first column pointer through that pointed to by the second column pointer.

The COLUMN command is provided to allow the user to specify the values of both the first and second column pointers. Because of the restriction on line sizes in MTS files (255 characters), the column pointers are restricted to a range of values from 1 to 255. The first column pointer must be less than or equal to the value of the second

pointer. The values assigned to the column range pointers place no additional restriction on the size of the individual lines within the files.

Regions

Some of the edit commands may perform their function on more than a single line of the file. When this is the case, the range of file lines upon which they operate may be specified by a name rather than always by the first and last line numbers. This name is specified by

/xxxxx

where "xxxxx" is from 0 to 7 non-blank characters, and is known as a line region name.

Edit commands for which a line region specification is valid require the name of the region as a parameter. Unlike the column range for which there is only one specification, several line region names may be defined. The REGION command is provided for defining and establishing the values of line regions. The interval over which the line region applies is defined as the line pointed to by the first value through the line pointed to by the second value. The values of the line pointers in a line region specification are any valid MTS line numbers, that is, any "n" where

$$-99999.999 \leq n \leq 99999.999$$

The restriction that the numerical value of the first pointer must be less than or equal to that of the second pointer applies. The value assigned to a region name places no additional restriction on the extent of MTS line numbers.

The editor has one predefined region name "/FILE" which is used to refer to the entire file.

Switches

Two general classes of switches are defined:

(1) Global switches which are set by default or by the user with the SET command and which govern the action taken by commands and error conditions which occur during execution. These are defined and explained under the SET edit command description below.

(2) Xec switches which are set as a result of some action occurring during execution. These are explained under the Xec procedure section below.

Checkpoint/Restore Facility

This facility allows the user to preserve the state of his file at any time. This is done by using the CHECKPOINT command to initiate a checkpoint buffer; then any commands which are executed that cause changes to the text of the file are recorded in the checkpoint buffer along with sufficient information to restore the changed lines to their state before the CHECKPOINT command was issued. The checkpointing feature may be suppressed for the duration of a command by the @NCH modifier.

The user may restore at any time all or portions of his file to the state it was in at the time he used the CHECKPOINT command by using the RESTORE command. The RESTORE command may be used any number of times, but it always restores the file to the state in which it was last checkpointed.

The state in which the file is to be checkpointed can only be changed by the use of the CHECKPOINT command which reinitiates the checkpoint buffer.

After initiating checkpointing with the CHECKPOINT command, an alternate mode of operation is to set the checkpoint switch off using the SET command; then, only those commands appended with a @CH modifier are checkpointed.

The editor works with only one copy of the file being edited and changes are made as they are entered. This is in contrast with editors run on other systems which work with two copies of the file and only replace the original with the edited version when requested.

Attention Interrupts

Issuing one attention interrupt causes the current command to be terminated immediately and the editor to return to edit command mode. A second attention interrupt without an intervening command causes the editor to return to MTS by calling the subroutine MTS. The editor may then be reentered by issuing the EDIT command.

Xec Procedures

An Xec is a sequence of commands stored as a named program which may be executed under the control of a count parameter (specifying how many times the sequence of commands is to be executed), the GOTO command, or a combination of these.

The XEC command is used to name a new Xec and to enter the commands to be stored. If the user already has an Xec named $name, the message

```
ALREADY DEFINED - ENTER "Y" TO REPLACE, "N" TO RETAIN
```

is typed and appropriate action is taken depending upon the reply. Otherwise, the user is prompted with a "?" to enter the command lines to comprise the Xec. Command lines consist of an optional label field followed by a command. The label field consists of 1 through 8 non-blank characters immediately followed by a colon (":"). Because of the format of the label field,

```
REP: PRI /A:
```

is a labelled command to print the region named /A:, while

```
REP :PRI /A:
```

is an unlabelled REPLACE command. Each command is checked for syntactic correctness as it is entered. If a command is found to contain a syntactic error, the appropriate error message is printed followed by

```
ENTER NEW COMMAND OR ":"  TO RETAIN
```

and the command is either discarded or retained accordingly. The ability to retain commands found to contain syntactic errors is provided so that one can, for example, refer to a region name in defining an Xec before defining the region name. In any case, if there remain syntactic errors when the Xec is executed, they are caught then. Entering of commands is terminated by entering either the pseudo-command END, a null line, or an end-of-file.

Three Xec switch values are defined. Two of these are SUCCESS, abbreviated S, and FAILURE, abbreviated F. They are complementary. They refer to the success or failure of a command which does some operation involving a form of search. Success implies that the command was successful at least once within its specified line region. Failure implies that it exceeded its line region or encountered an end-of-file before fulfilling its function. The third form of Xec switch is ENDOFFILE, abbreviated EOF or E. This value is either ON or OFF. It is ON if an end-of-file occurs during a read or write operation. It is OFF if the read or write operation was completed without an end-of-file. The Xec switch value may be tested by the GOTO command.

COMMAND NAMES

As in the MTS command language, each command is recognized by its minimum distinguishable abbreviation, usually one or two letters. A few commands begin with non-alphabetic characters and are recognized by the presence of these characters. The command names and their abbreviations are as follows:

ALTER	A
BLANK	B
CHANGE	C
CHECKPOINT	CHE
COLUMN	COL
COPY	CO
DELETE	D
DOCUMENT	DO
EDIT	ED
EXPLAIN	E
GOTO	G
INSERT	I
LINE	L
MATCH	M
MTS	MT
OVERLAY	O
PRINT	P
REGION	REG
RENUMBER	REN
REPLACE	R
RESTORE	RES
SCAN	S
SET	SE
SHIFT	SH
STOP	ST
XEC	X
+n	n
-n	-n
$name	$name
/name	/name

where "n" is an integer and "name" is a sequence of 0 through 7 non-blank characters.

There are four general classes of commands:

(1) those starting with an alphabetic keyword,
(2) two commands which are a signed integer,
(3) those prefixed by a "$", and
(4) those prefixed by a "/".

All commands starting with an alphabetic keyword require parameters. In many cases if the command is issued without parameters, one or more of several default values are assigned to the missing parameters. Many of these commands may have modifiers appended to the keyword portion. These modifiers are fully described below along with all forms of parameters.

The two commands consisting of signed integers are used to control the value of the current line number.

The commands which are names prefixed by a "$" are lists of Xec commands prepared by the user. They are executed upon the use of their name, which

has been previously assigned to them. The name is defined by the user along with the list of commands to be used in conjunction with it.

The prefix "/" followed by a name causes the line number values of the line region represented by that name to be printed provided that such a name has been previously defined by the user.

COMMAND MODIFIERS

Command Modifiers are composed of an at-sign "@" followed by one to three characters. Each modifier must be appended to the command which it modifies without intervening blanks. More than one such modifier may be appended to a command name. For example,

```
PRINT@NL@X 100 200
```

causes lines 100 thru 200 to be printed in hexadecimal without line numbers.

Two classes of modifiers are defined:

(1) Modifiers which override permanent switch assignments for the duration of the command to which they are appended. See the SET edit command for an explanation of these switch settings.

@CH	Turn the checkpoint switch on for this command.
@NCH	Turn the checkpoint switch off for this command.
@L	Turn line-number prefixing on for this command.
@NL	Turn line-number prefixing off for this command.
@V	Turn verification on for this command.
@NV	Turn verification off for this command.
@X	Turn hexadecimal conversion on for this command.
@NX	Turn hexadecimal conversion off for this command.

(2) Modifiers which control the action performed by the command.

@A	Apply the given command throughout the given line range, rather than only once.
@PC	When using the SCAN and ALTER (CHANGE) commands, the cclumn number of the column where the string found or altered begins is printed.

COMMAND PARAMETERS

Strings

A string is a sequence of characters delimited at either end by a current delimiter character which may be any one of the following characters:

' " ¢ < > ( ) , ? : ; % & | ¬ _ ! # @ =

The first character from this set encountered when scanning for a string is taken as the current delimiter character for that string. If a command requires a pair of strings, they must be presented as the delimiter, followed by the first sequence of characters, followed by the delimiter, followed by the second sequence of characters, followed by a third delimiter. If hexadecimal conversion is on, the character string is inspected, and, if it is found to contain only legal hexadecimal characters, it is converted to internal character form.

Examples:

The character string:	'STRING'
The null string:	""
The hexadecimal string:	#E1E2D9C9D5C7#
A pair of strings:	'ABC'DEF'

If the final delimiter is omitted when entering a string or pair of strings, the editor types

```
: FINAL DELIMITER? :
```

and waits for an answer to be typed after the second colon. If the correct final delimiter is typed followed by ending the line, the string or strings are accepted; otherwise, the command is cancelled.

Line Numbers

A file editor line number may be any legal MTS line number or one of

*	The current line
*F	The first line of the file
*L	The last line of the file

The file editor maintains a current line pointer which is initially set to the first line of the file and is modified by the LINE, MATCH, SCAN, n, +n, and -n commands.

Counts

A count parameter is an integer or (in some commands) a keyword expression of the form

COUNT=integer

or

C=integer

Region Names

A region name is a slash "/" followed by 0 through 7 non-blank characters. Region names are defined by the REGION command and denote line number ranges in a file. There is one predefined region, namely "/FILE" which is equated to the range -99999.999 through 99999.999, that is, all of the MTS file.

Xec Names

An Xec name is a dollar sign "$" followed by 0 through 7 non-blank characters. Xec names are defined by the XEC command and denote series of stored commands which may be executed as stored programs.

Keyword Parameters

A keyword parameter is either a literal series of characters or a literal series of characters followed by an equal sign "=" followed by a value field.

Examples:

```
COLUMNS
REGIONS
ECHO=ON
```

EDIT COMMAND DEFINITIONS

In the following command descriptions, the symbol "lpar" is used to denote any of the following combinations of parameters:

/name	the line range assigned to the region name
linenumber	the given line number

linenumber1 linenumber2	the given range of line numbers
linenumber COUNT=integer	"integer" lines beginning with the given line number (COUNT may be abbreviated by C)
COUNT=integer	"integer" lines beginning with the current line
(nothing)	the current line, except in the SCAN and MATCH commands, for which the default line number range is the current line through the last line of the file

The following notation conventions are used in the prototypes of the commands:

- lower case - represents a generic type which is to be replaced by an item supplied by the user.
- upper case - indicates material to be repeated verbatim in the command.
- brackets [ ] - indicate that material within the brackets is optional.
- braces {} - indicate that the material within the braces represents choices, from which exactly one must be selected. The choices are separated by vertical bars.
- underlining - indicates the minimum abbreviated form of the command or parameter. Longer abbreviations are acceptable.

The following pages give a complete summary of the commands in the edit command language.

Summary of Edit Command Prototypes

Commands	Parameters	Modifiers and Xec Switches								
		A	CH	L	PC	V	X	S	F	EOF
ALTER	lpar ['str1'str2']	x	x	x	x	x	x	x	x	x
BLANK	lpar ['str']		x	x		x				x
CHANGE	lpar ['str1'str2']	x	x	x	x	x	x	x	x	x
CHECKPOINT	[OFF]									
COLUMN	[m [n]]									
COPY	lpar [TO [linenumber]]		x	x		x				
DELETE	lpar		x							x
DOCUMENT	lpar									x
EDIT	filename									x
EXPLAIN	[{command|ERROR}]									
GOTO	label [[ON] condition]									
INSERT	[linenumber]['str']		x	x		x	x			
LINE	linenumber			x		x	x			x
MATCH	lpar ['str']	x		x		x	x	x	x	x
MTS	[command]									
OVERLAY	lpar ['str']		x	x		x				x
PRINT	{lpar [check]|keyword|$name}			x			x			
REGION	/name [line1 [line2]]									
RENUMBER	[linenumber [increment]]		x							
REPLACE	lpar ['str']		x	x		x	x			x
RESTORE	lpar			x		x	x			
SCAN	lpar ['str']	x		x	x	x	x	x	x	x
SET	keyword=value									
SHIFT	lpar {LEFT|RIGHT} count		x	x		x	x			x
STOP										
XEC	$name									
+n				x		x	x			x
-n				x		x	x			x
$name										
/name										

April 1974

ALTER

EDIT COMMAND DESCRIPTION

Prototype: i) ALTER lpar 'string1'string2'

ii) ALTER lpar

Action: This command causes the first occurrence (or all occurrences if the @A modifier appears) of "string1" in the given line number and column range to be replaced by "string2". If no strings are supplied, the pair of strings given in the preceding ALTER with strings specified is used.

Modifiers: @A, @CH, @NCH, @PC, @V, @NV, @X, @NX, @L, @NL

Xec Settings: ALTER succeeds if "string1" is found at least once and fails otherwise. The end-of-file switch is turned on if the end-of-file is encountered.

Examples: ALTER 'IT'IS'

causes the first occurrence of IT in the given column range of the current line to be altered to IS.

A@A@X /FILE 'C1'E2'

causes all occurrences of the hexadecimal string C1 (the character A) in the given column range of the entire file to be altered to hexadecimal E2 (character S).

BLANK

EDIT COMMAND DESCRIPTION

Prototype: i) BLANK lpar 'string'

ii) BLANK lpar

Action: If a string is given, this command causes the given line number range to be blanked according to that string, that is, positions in the line corresponding to blanks in the string are replaced with blanks; positions corresponding to non-blank characters are not changed. If no string is given, each line of the given range is typed on the user's terminal and he is then prompted (with a "?") to enter a line to be used as the blanking pattern for the line typed.

Modifiers: @CH, @NCH, @V, @NV, @L, @NL

Xec Settings: The end-of-file switch is turned on if the end-of-file is encountered.

Example: B * 'AAA '

causes the fourth through sixth characters of the current line to be replaced by blanks.

CHANGE

EDIT COMMAND DESCRIPTION

Prototype:

i) CHANGE lpar 'string1'string2'

ii) CHANGE lpar

Action: The CHANGE command is a synonym for the ALTER command. See the ALTER command description above.

CHECKPOINT

EDIT COMMAND DESCRIPTION

Prototype: i) CHECKPOINT

ii) CHECKPOINT OFF

Action: The checkpoint/restore feature provides the capability to establish a base or checkpoint in editing to which all or any part of the file may be restored (see the RESTORE command).

CHECKPOINT with no parameter causes the current checkpoint buffer to be reinitialized or, if it is the first instance of the CHECKPOINT command, causes a one page buffer to be obtained and initialized (additional buffers are obtained as needed). CHECKPOINT OFF causes the current buffer to be released and the checkpoint feature to be disabled.

Modifiers: None.

Xec Settings: None.

April 1974

COLUMN

EDIT COMMAND DESCRIPTION

Prototype:

i) COLUMN m n

ii) COLUMN m

iii) COLUMN

where "m" and "n" are positive integers.

Action: The first form of this command sets the column pointers to the "m" and "n"th columns, respectively. The second form sets the pointers to "m" and 255, and the third resets them to their initial values, 1 and 255. The column pointers are used by the ALTER, BLANK, MATCH, OVERLAY, SCAN, and SHIFT commands as limits for editing in any given line. The column range is defined as extending from the character pointed to by the first column pointer through that pointed to by the second pointer.

Modifiers: None.

Xec Settings: None.

COPY

EDIT COMMAND DESCRIPTION

Prototype: i) COPY lpar [TO] linenumber

ii) COPY lpar [TO]

Action: This command causes the specified line number range to be copied (without being deleted) into the file immediately between the line whose line number is given in the TO field and the following line; the second form defaults the TO field to the current line. Note that if lowercase conversion is enabled, the word "to" (if used) must be in upper case.

Note that the ambiguous command

```
COPY linenumber1 linenumber2
```

is interpreted as meaning

```
COPY linenumber1 linenumber2 TO *
```

not

```
COPY linenumber1 TO linenumber2
```

Modifiers: @CH, @NCH, @V, @NV, @L, @NL

Xec Settings: None.

April 1974

DELETE

EDIT COMMAND DESCRIPTION

Prototype: DELETE lpar

Action: This command causes the lines of the given line number range to be deleted from the file.

If the user intends to delete a large set of contiguous lines from his file, he should use the MTS COPY command to copy the retained portions of the file to a temporary file. The original file can then be emptied and the temporary version can be copied back. This method is more efficient.

Modifiers: @CH, @NCH

Xec Settings: The end-of-file switch is turned on if the end-of-file is encountered.

DOCUMENT

EDIT COMMAND DESCRIPTION

Prototype: DOCUMENT lpar

Action: The DOCUMENT command is intended to be used in adding comments to assembly language programs. It causes each line of the given line number range to be typed out, followed by the editor's issuing a "?" to prompt the user to enter a comment to be appended to the line typed out. The comment is placed at the location marked by the first column pointer, or if the text already extends beyond the first column pointer, after the text with one intervening blank. If the comment extends the line beyond column 70, the comment is broken at a blank (or if it contains no blanks, at column 70) and continued on a line inserted after the line typed out.

Modifiers: None.

Xec Settings: The end-of-file switch is turned on if the end-of-file is encountered during the documenting operation.

Example:

```
COL 28
DOC *
LABELA   AH    2,B
?NOW ADD B TO COMPUTE NEW INCREMENT FOR LINE NUMBER POINTER
PRI@NL
LABELA   AH    2,B          NOW ADD B TO COMPUTE NEW INCREMENT
+1@NV
PRI@NL
*                           FOR LINE NUMBER POINTER
```

April 1974

EDIT

EDIT COMMAND DESCRIPTION

Prototype: EDIT filename

Action: This command causes the editor to release the file currently being edited and to obtain the specified file for editing. Line number ranges have no effect. Explicit concatenation may not be specified. The file editor operates with implicit concatenation off.

If checkpointing is enabled, it remains enabled, but the checkpoint status is reset to one empty buffer, that is, the effect is the same as if

```
CHECKPOINT OFF
EDIT filename
CHECKPOINT
```

had been specified.

If EDITAFD is ON, "filename" becomes the currently active file (*AFD*).

Modifiers None.

Xec Settings: The end-of-file switch is turned on if the new file is empty.

EXPLAIN

EDIT COMMAND DESCRIPTION

Prototype:

i) EXPLAIN

ii) EXPLAIN command

iii) EXPLAIN ERROR

Action: The EXPLAIN command provides information about the syntax and operation of file editor commands. The first form provides general information. The second form provides information about the specified command. The third form is used when the BRIEF switch is enabled (see the SET command) to explain the last preceding error comment.

Modifiers: None.

Xec Settings: None.

April 1974

GOTO

EDIT COMMAND DESCRIPTION

Prototype: i) GOTO label

ii) GOTO label [ON] condition

where "condition" is any of SUCCESS, S, FAILURE, F, ENDOFFILE, EOF, or E.

Action: The GOTO command is used inside Xecs to alter the sequential interpretation of commands. The first form specifies an unconditional transfer to the command with the given label; the second form specifies a transfer if the specified condition is met. The label may be one to eight non-blank characters. There are two predefined labels, namely COUNT and END. COUNT causes the interpretation count to be decremented by one and interpretation to be continued from the beginning of the Xec. END causes the Xec to be terminated immediately.

Modifiers: None.

Xec Settings: None.

Examples: See the XEC command for examples.

INSERT

EDIT COMMAND DESCRIPTION

Prototype:

i) INSERT linenumber 'string'

ii) INSERT 'string'

iii) INSERT linenumber

iv) INSERT

Action: The INSERT command with a string specified causes that string to be inserted as a line into the file between the line with the given line number (which defaults to the current line if no line number is specified) and the succeeding line. If no string is specified, the editor enters fast insertion mode. In this mode of operation, all lines entered up to a null line or an end-of-file are inserted between the specified (or defaulted) line and the following line. The line numbers are calculated by an algorithm which varies the increment depending on the difference between the line number of the last line inserted and the following line.

If the given line number does not correspond to any line in the file, the first line is inserted there.

Modifiers: @CH, @NCH, @V, @NV, @X, @NX, @L, @NL

Xec Settings: None.

Examples: IN 'NEW LINE'

causes the line

NEW LINE

to be inserted after the current line.

```
:i 320
?line 1
?line 2
?line 3
?$ENDFILE
```

inserts the given three lines after line 320.

April 1974

LINE

EDIT COMMAND DESCRIPTION

Prototype: LINE linenumber

Action: The LINE command causes the given line number to become the current line number.

Modifiers: @V, @NV, @X, @NX, @L, @NL

Xec Settings: The LINE command turns the end-of-file switch on if no line with the given line number is found in the file.

MATCH

EDIT COMMAND DESCRIPTION

Prototype: i) MATCH lpar 'string'

ii) MATCH lpar

Action: MATCH causes the editor to search the given line number range of the file for a line that matches the given string starting at the left column position. Positions in which the current fill character (see the SET command) occur in the string match any character in the corresponding position in the column range. If no string is given, the string given in the previous MATCH with a string is used.

Modifiers: @A, @V, @NV, @X, @NX, @L, @NL

Xec Settings: MATCH succeeds if the given string is found at least once, and fails otherwise. The end-of-file switch is turned on if the end-of-file is encountered during matching.

Example:

```
COL 10 15
MATCH@A /FILE 'BAL   '
```

This example of the MATCH command can be used to match all occurrences of the BAL or BALR instruction in an assembly language program.

April 1974

MTS

EDIT COMMAND DESCRIPTION

Prototype: i) MTS

ii) MTS command

Action: The first form of this command returns control to the MTS command level. The second form returns control to MTS and issues the MTS command given in the parameter field. In either case, the editor may be reentered by issuing the $EDIT command.

All Xecs and line number regions remain defined. All switch values changed with the SET edit command remain in effect.

The name of the file being edited is saved for use with a future $EDIT command. See the $EDIT MTS command description for details.

Modifiers: None.

Xec Settings: None.

Example: MTS $LIST FILEA

causes control to be returned to MTS and the file FILEA to be listed.

OVERLAY

EDIT COMMAND DESCRIPTION

Prototype: i) OVERLAY lpar 'string'

ii) OVERLAY lpar

Action: The first form causes the specified line number range to be overlaid with the given string. The second form causes the given column range of each line to be printed followed by the editor issuing a "?" as the prompting character and awaiting a line entered by the user to be used to overlay the line typed. OVERLAY causes characters in the string or line entered to replace the characters in the corresponding positions in the column range. Where the fill character occurs in the string or the line entered, the coresponding character in the line overlaid is left unchanged. The initial fill character is the blank. It may be changed with the SET command.

Modifiers: @CH, @NCH, @V, @NV, @L, @NL

Xec Settings: The end-of-file switch is turned on if the end-of-file is encountered during the overlaying operation.

Example: If the column pointers are set to 1 and 255 and the current line is

```
THEN AND THERE
```

then the command

```
OVERLAY 'NOW '
```

causes the current line to become

```
NOWN AND THERE
```

while

```
SET FILL=%
OVERLAY 'NOW '
```

causes it to beccme

```
NOW  AND THERE
```

April 1974

PRINT

EDIT COMMAND DESCRIPTION

Prototype:

i) PRINT lpar

ii) PRINT lpar check

iii) PRINT keyword

iv) PRINT $name

where "check" is one of CHECKPOINTED, CHECKPOINT, CHECK, or CH and "keyword" is one of COLUMNS, C, FILE, F, REGIONS, R, XECS, or X.

Action:

"PRINT lpar" causes the lines of the given line number range to be printed.

"PRINT lpar check" causes the lines in the given line number range which have been changed since the last CHECKPOINT command to be printed in their checkpointed forms.

"PRINT COLUMNS" or "PRINT C" causes the values of the current column pointers to be printed.

"PRINT FILE" or "PRINT F" causes the name of the file being edited to be printed.

"PRINT REGIONS" or "PRINT R" causes the names of the currently defined regions to be printed.

"PRINT XECS" or "PRINT X" causes the names of the currently defined Xecs to be printed.

"PRINT $name" causes the commands in the Xec named $name to be printed.

Modifiers:

"PRINT lpar" and "PRINT lpar check": @L, @NL, @X, @NX
Other forms: None.

Xec Settings:

None.

Examples:

P@X@L /FILE

causes the entire file to be printed in hexadecimal with line number prefixing.

```
P $A
```

causes the contents of the Xec named $A to be printed.

April 1974

REGION

EDIT COMMAND DESCRIPTION

Prototype:

i) REGION /name linenumber1 linenumber2

ii) REGION /name linenumber

iii) REGION /name

where "/name" is a region name.

Action: The REGION command defines the line number range to be associated with a region name, either "linenumber1" through "linenumber2", "linenumber" alone, or the current line number, respectively.

Modifiers: None.

Xec Settings: None.

Example: REG /A

causes the current linenumber to be associated with the region name /A.

RENUMBER

EDIT COMMAND DESCRIPTION

Prototype: RENUMBER linenumber increment

Action: The RENUMBER command causes the entire file to be renumbered beginning with "linenumber" and increasing by "increment". The "linenumber" and "increment" parameters both default to 1 if not specified and both may assume positive, negative, and zero values.

The following system or user actions may cause either the entire contents or part of the contents of the file to be lost:

1) any action which results in the user being signed off, such as system failure or time limit exceeded,

2) any combination of "linenumber" and "increment" which causes a line number greater than 99999.999 or less than -99999.999, or

3) an attention interrupt issued during execution of this command.

These problems are created by the method used to renumber the file. The editor copies the entire contents of the file to a file named "-EDQQSV", empties the file being edited, and copies the contents back. If the file has been copied to -EDQQSV and any of the above three conditions occur, a part or all of the file contents may be lost.

Modifiers: @CH, @NCH

Xec Settings: None.

April 1974

REPLACE

EDIT COMMAND DESCRIPTION

Prototype: i) REPLACE lpar 'string'

ii) REPLACE lpar

Action: REPLACE causes the lines in the line number range to be replaced by the given string. If no string is given, each line is typed and the user is prompted with a "?" to enter a replacement line.

Modifiers: @CH, @NCH, @V, @NV, @X, @NX, @L, @NL

Xec Settings: The end-of-file switch is turned on if the end-of-file is encountered during the replacing of lines.

RESTORE

EDIT COMMAND DESCRIPTION

Prototype: RESTORE lpar

Action: RESTORE causes the lines in the given line number range to be restored to their condition at the time of the last CHECKPOINT command if the checkpoint/restore feature is enabled.

Note that

```
RESTORE
```

is the same as

```
RESTORE *
```

and only restores the current line. If the entire file is to be restored to the condition before the last CHECKPOINT command,

```
RESTORE /FILE
```

should be specified.

Modifiers: @V, @NV, @X, @NX, @L, @NL

Xec Settings: None.

SCAN

EDIT COMMAND DESCRIPTION

Prototype: i) SCAN lpar 'string'

ii) SCAN lpar

Action: SCAN causes the editor to search the current column range of the given line number range for the first occurrence (or, if the @A modifier appears, all occurrences) of the given string. If no string is given, the string given in the previous SCAN command with a string is used. The last line found containing the string becomes the current line, and, if the @PC modifier appears in the command, the column at which the first character of each occurrence of the string begins is printed.

Modifiers: @A, @PC, @V, @NV, @X, @NX, @L, @NL

Xec Settings: SCAN succeeds if the given string is found at least once and fails otherwise. The end-of-file switch is turned on if the end-of-file is encountered during scanning.

Example:
```
COL 10 15
SCAN@A@PC /FILE 'BALR'
```

finds all occurrences of the BALR instruction in a standard-format 360 assembler program.

SET

EDIT COMMAND DESCRIPTION

Prototype: SET keyword=value

Action: The SET command is used to alter the values of a number of switches which govern the editing process and to change the fill character.

The switch keywords may take the values ON or OFF. The keywords, their abbreviations, and their meanings are as follows:

BRIEF	B	When BRIEF is ON, errors are signaled by the message "ERROR" and the complete message may be obtained by issuing the command "EXPLAIN ERROR". Default: OFF.
CC		When CC is ON, lines from the file are printed with carriage control. Default: OFF.
CHECKPOINT	CHECK C	If a checkpoint buffer has been allocated and the CHECKPOINT switch is ON, changes to the file are recorded by the checkpoint/restore feature (see the CHECKPOINT and RESTORE commands). Default: ON.
ECHO	E	When ECHO is ON, commands entered from the source stream and those read from Xecs are echoed on SPRINT. Default: ON (batch), OFF (terminal).
ERROREXIT	ERROR ERR ER	When ERROREXIT is ON and an error occurs, the editor terminates execution by calling ERROR after printing the appropriate error message. Default: ON (batch), OFF (terminal).
FILL	F	The fill character is set to the single character following the equal sign of the keyword expression. The fill character is used in the MATCH and OVERLAY commands. Default: blank

HEX	H X	When HEX is ON, input strings are converted from hexadecimal to internal character form on input, and output lines are printed in hexadecimal format. Default: OFF.
LINENUMBER	LNR L	When LINENUMBER is ON, the line number of each output line is printed before the line. Default: ON.
TRIM	T	When TRIM is ON, lines written into and read from the file are trimmed of trailing blanks. Default: OFF.
VERIFY	V	When VERIFY is ON, edited lines are printed following the execution of the ALTER, BLANK, CHANGE, COPY, INSERT (only the form with a string), LINE, MATCH, OVERLAY, REPLACE, RESTORE, SCAN, SHIFT, -n, and +n commands. Default: ON.

Modifiers: None.

Xec Settings: None.

Examples:

```
SET VERIFY=ON

SET FILL=$
```

SHIFT

EDIT COMMAND DESCRIPTION

Prototype: SHIFT lpar direction count

where "direction" is LEFT or L, RIGHT or R and "count" is an integer satisfying 1 ≤ "count" ≤ 255.

Action: SHIFT causes the contents of the given column range of the given line number range to be shifted in the given direction by "count" characters. Blanks are shifted into the positions vacated.

Modifiers: @CH, @NCH, @V, @NV, @X, @NX, @L, @NL

Xec Settings: The end-of-file switch is turned on if the end-of-file is encountered during shifting.

Example:

```
COL 10 15
SHIFT * LEFT 2
```

would change

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

to

```
ABCDEFGHILMNO  PQRSTUVWXYZ
```

April 1974

STOP

EDIT COMMAND DESCRIPTION

Prototype: STOP

Action: STOP causes the file editor to return to MTS and to release its control storage. Thus, the next EDIT command enters the editor with the default switch settings, column range, and regions (/FILE, only), and with no XECs defined. This is the same as if the editor had not been used before. See the $EDIT MTS command for details.

Modifiers: None.

Xec Settings: None.

XEC

EDIT COMMAND DESCRIPTION

Prototype: XEC $name

where "$name" is an Xec name.

Action: A new Xec name is created. All the commands entered until the next null line, end-of-file, or END command become the text of the Xec. See the discussion of Xec procedures above for the details of using Xec definitons.

Modifiers: None.

Xec Settings: Not applicable.

Example: The following Xec could be used to print the first ESD line and the END line of each object module in a file:

```
XEC $A
LINE *F
A:MAT * *L ' ESD'
GOTO END ON EOF
MAT * *L ' END'
GOTO A
END
```

The following Xec could be used to locate all lines that have either an "&" or a "." in column 1 (or whichever is selected by the column range).

```
XEC $B
LOOP:  M * * '&'
M * * '.'
TONV
GOTO END ON EOF
GOTO LOOP
END
```

+n, -n

EDIT COMMAND DESCRIPTION

Prototype:

i) n

ii) +n

iii) -n

where "n" is an unsigned integer.

Action: This command causes the current line pointer to be moved forward or backward by "n" lines. The sign of the number indicates the direction to move.

Thus, if the current line is 4, the command

```
1
```

moves to the next line (which is not necessarily 5) and subsequently

```
-2
```

moves to the line preceding line number 4 (which is not necessarily 3).

Modifiers: @V, @NV, @X, @NX, @L, @NL

Xec Settings: The end-of-file switch is turned on if the end-of-file is encountered in either direction.

$name

EDIT COMMAND DESCRIPTION

Prototype: i) $name count

ii) $name

where "$name" is an Xec name and "count" is a positive integer.

Action: This command initiates the execution of the Xec named "$name". If no count is given, it is assumed to be one. If the Xec does not contain GOTO commands, the sequence of commands is executed "count" times. If the Xec contains GOTOs, the sequence of control and the number of times each command is executed is determined by the interaction of the count and the GOTO commands.

Modifiers: None.

Xec Settings: None.

April 1974

/name

EDIT COMMAND DESCRIPTION

Prototype: /name

where "/name" is the name of a region.

Action: This command causes the line numbers which are the bounds of the given region to be printed.

Modifiers: None.

Xec Settings: None.

Example: /FILE

causes the line numbers

-99999.999 99999.999

to be printed.

April 1974

DEBUG MODE

The Symbolic Debugging System (SDS) is a conversational facility for testing and debugging programs. This facility was originally provided only for 360-assembler language programs, but it has now been extended to include FORTRAN programs. SDS enables the user to initiate the execution of a program and monitor its performance by displaying or modifying instructions and data at strategic points in the program.

SDS may be invoked in an active manner with the MTS command

```
$DEBUG FDname
```

where FDname is the file(s) containing the program to be debugged. The parameters to the $DEBUG command are the same as those for the $RUN command; hence, if the program refers to logical I/O units or has a system parameter field, these also may be included on the $DEBUG command. For example,

```
$DEBUG PROGRAM SCARDS=INPUT SPRINT=OUTPUT PAR=EXEC
```

The user communicates with SDS by entering debug commands from his terminal. SDS signals its readiness to accept a command by printing the prefix character "+" in column one. This prefix character precedes all SDS messages and diagnostics. Initially, commands to SDS are read from the pseudo-device *SOURCE* and SDS output messages and error diagnostics are written on *SINK*. Commands may be entered in either the upper or lower case.

When the program has been successfully loaded, SDS prints

```
+READY
+
```

at which point SDS is ready to accept its first debug command. A command is a request that SDS do something with the program being debugged. For instance, the RUN command initiates execution of the program, and the MAP command lists relevant information about each loaded section. The BREAK command is provided to insert breakpoints at strategic locations in the program. If a program interrupt occurs during execution, or if a breakpoint is encountered, control is returned to SDS which explains why the program has stopped running and requests another command. The DISPLAY command may then be used to selectively examine specific locations in memory or the contents of the general and floating-point registers. Changes to data and to instructions may be made with the MODIFY command. The program may be restarted with the CONTINUE command. In this manner, the user may "step" through the program and monitor its performance. A complete description of the debug command language appears at the end of this section.

Logical I/O unit assignments and the value of the system parameter list may be specified on the $DEBUG command in the same format as the $RUN command.

If the user has omitted these assignments from the $DEBUG command, or desires to change or reset any of the assignments, he may do so with the SET debug command. For example, after issuing the MTS command

```
$DEBUG -LOAD
```

the debug command

```
SET 5=DATA 6=RESULTS PAR=LIST
```

assigns the files DATA and RESULTS to logical I/O units 5 and 6, respectively, and sets the system parameter list to the character string LIST. This is the same as if the MTS command

```
$DEBUG -LOAD 5=DATA 6=RESULTS PAR=LIST
```

had been issued originally.

Assigning logical I/O units with the SET command may be used to reposition files but does not affect the status of other types of pseudo-devices (such as *SOURCE* and the position of tapes). The PAR option must be the last parameter specified since the remainder of the input line is taken as the system parameter list.

An alternative method for invoking SDS is with the MTS command

```
$SET DEBUG=ON
```

This method uses SDS in a passive manner. All programs loaded after this point are processed by SDS. The MTS commands of $RUN and $START/$RESTART do not start execution of the program directly, but transfer control to SDS with an implicit debug command of RUN or CONTINUE (respectively). SDS intercepts any error that occurs, prints an error comment, and returns to MTS command mode. At this point the user, if he desires, may explicitly enter debug command mode with the MTS command

```
$SDS
```

to determine what went wrong with his program. This mode of operation may be disabled by the MTS command

```
$SET DEBUG=OFF
```

The default for the DEBUG option is OFF. When the DEBUG option is ON, all user programs, language translators, and other publicly available programs are processed by SDS; hence, the user should be aware of which programs he does and does not want SDS to process. Normally, when the user is not running one of his own programs, the DEBUG option should be OFF.

DISPLAY AND MODIFY PROCESSING

SDS Parameters

The most useful facility of SDS is the ability to selectively refer to locations in the program being debugged. This may be done symbolically, by relative address, or by absolute (virtual) address. In the symbolic mode, a location is referred to by the symbol assigned to that location at program translation time.

Symbolic referencing is possible only with those language processors which generate a symbol table with the object programs they produce. Currently, this is done by the 360 Level G Assembler (*ASMG) and the FORTRAN G compiler (*FTN). With *ASMG and *FTN, the TEST parameter must be specified for the processor to produce symbol table information. The formats are:

```
$RUN *ASMG SCARDS=source SPUNCH=object PAR=TEST

$RUN *FTN PAR=SOURCE=source,LOAD=object,TEST
```

When referencing by relative or virtual addresses, SDS processes all loaded programs which were produced by a language translator that generates an object module. This excludes interpretive translators such as WATFOR and SNOBOL4.

The general format for an SDS parameter is

S(i)±j

"S" may be

(1) a symbolic location appearing in a source language program,

(2) a relative address assigned at assembly or compilation time, or

(3) an absolute (virtual) address corresponding to a location within the loaded program.

"i", the index, may be a signed or unsigned decimal integer which represents the ith translated element (that is, an instruction, or data item) starting with or preceding S. S(0) and S(1) are identical to S. Recursively, "i" may also be an entire SDS parameter of the form S(i)±j. For FORTRAN arrays, "i" may be a series of indices, separated by commas.

"j", the displacement, may be an unsigned hexadecimal integer which represents the number of bytes following or preceding the location specified by S(i).

An SDS parameter is terminated by a blank or by a zero-level comma (a comma which is not contained in a parenthesized expression).

The SDS indexing mechanism ignores zero duplication factors and assembler generated spaces. In the 360 assembler example below, HW(2) refers to the same location as CH; HW(3) and CH(2) both refer to the same location as FW.

```
000020  0009              HW      DC    H'9'
000028                            DS    0D
000028  C1C2C3C4C540      CH      DC    C'ABCDE '
000030  00000005          FW      DC    F'5'
```

Illegal operation codes do not terminate SDS indexing operations. The instruction is assumed to be 2 bytes in length (RR type) and indexing proceeds.

Both the index and the displacement are optional. Examples of legal SDS parameters are:

```
ALPHA
ALPHA(2)
ALPHA(-3)
BETA+20
BETA(6)+4
BETA(8)-C
```

Recursively, "i" may be an entire SDS parameter of the form S(i)±j. The index "i" is assumed to be a decimal integer unless it is a symbol or a relative or absolute address (with the @R or @A modifiers appended). When the index is a symbol or an address, the contents of the location specified is used as the index for the indexing operation. As many levels of recursion as desired may be specified, for example,

```
A(I)
B(A(I))
C(B(A(3)))
D(20@R)
```

For FORTRAN arrays, "i" may be a series of indices separated by commas. Up to seven indices may be specified, corresponding to the number of dimensions in the array. Alternatively, any FORTRAN array may be referred to linearly (with one index). For example,

```
A(6)
B(4,3)
C(1,2,3,4,5,6,7)
```

Symbolic indices may also be used, for example,

```
A(I)
B(J,K)
```

To refer to statement numbers in FORTRAN programs, a prefix must be used to distinguish the statement number from a relative address. The "#" must prefix the statement number if it is an external (user-defined) statement number. For example,

#10

is the user-defined statement number 10. "IS#" must prefix the statement number if it is an internal (source-listing) statement number. For example,

IS#10

is the source-listing statement number 10. Only those statement numbers which define executable FORTRAN statements may be used. An executable statement is defined as a statement which is from one of the following categories:

(1) Assignment statements
(2) Control statements
(3) I/O statements

All others, such as those defining DIMENSION, REAL, INTEGER, DATA, COMMON, SUBROUTINE, FUNCTION, ENTRY, EQUIVALENCE, and FORMAT statements are non-executable and therefore undefined. Both internal and external statement numbers must be specified without leading zeros.

It is frequently necessary to distinguish explicitly between a symbolic, a relative, or an absolute address. For example, "A14" could be interpreted as the symbol "A14", or the relative or absolute address A14 (hex). To indicate a relative or absolute address, the @R or @A modifiers may be appended to any parameter; hence, "A14@R" becomes the relative address A14 (hex). If the parameter name begins with a letter, it is assumed to be a symbol. Thus, "A14" is treated as a symbol, and to specify the relative location A14 either "0A14" or "A14@R" must be specified.

If the parameter is greater than or equal to 100000 (hex), the address is assumed to be absolute and the @A modifier need not appear. If the parameter is less than 100000, the address is assumed to be relative.

The @Q modifier forces any string of characters to be treated as a symbol.

SDS assumes all indices are decimal and all displacements are hexadecimal unless overridden by the @X or @D modifiers. The @X modifier specifies hexadecimal conversion of an integer constant, while the @D specifies decimal conversion of an integer constant. The @X and @D modifiers affect only the immediate numbers to which they are appended.

The following 360 assembler language program illustrates the use of SDS parameters.

```
000000                          PROG    CSECT
000000                                  USING *,15
000000  5810 F014                       L       1,ALPHA
000004  5A10 F018                AD     A       1,C
000008  5010 F014                       ST      1,ALPHA
00000C  58A0 F01C                       L       10,VSYS
000010  07FA                           BR      10
000014  0000000A                ALPHA  DC      F'10'
000018  00000014                C      DC      F'20'
00001C  00000000                VSYS   DC      V(SYSTEM)
000020                          GAMMA  DS      20F
000070  1C022C333C1C022C        DELTA  DC      2P'1,22,333'
                                        END
```

The following is a list of SDS parameters and the corresponding relative addresses.

Parameter	Address
PROG	000000
PROG+4	000004
PROG(7)	000018
AD	000004
AD(-1)	000000
AD+4	000008
C	000018
C(0)	000018
C(1)	000018
C+C	000024
VSYS	00001C
VSYS-10@X	00000C
VSYS-10	00000C
VSYS-10@D	000012
VSYS-A	000012
GAMMA	000020
GAMMA(20)	00006C
GAMMA(C)	00006C
DELTA(3)	000073
DELTA(-22)	000018
C@R	00000C
C@R(2)	000010
C@R-4	000008
C@R+C	000018
10(2)	000014
20+A	00002A
70	000070
70(2)	000071
70(3)	000073
70(4)	000075
70(5)	000076

To display the current value of a location, the DISPLAY command may be used. For example,

```
DISPLAY ALPHA
```

converts the location ALPHA according to its type and length and prints the result. The type refers to the format in which it is displayed (that is, fixed-point, floating-point, character, hexadecimal, etc.) and the length refers to the number of bytes printed. The ALPHA from the above program is displayed as

```
ALPHA  'F'  +10
```

Locations may also be referred to by the relative addresses assigned at translation time. To display the contents of the relative location C, the command

```
DISPLAY C@R
```

may be used. This displays the instruction in the form

```
AD(3)  'I'  L   A0F01C
```

The @R modifier is used to indicate a relative address. If no symbol table is provided at translation time, the location is printed in hexadecimal format as

```
00000C  'X'  58A0F01C
```

The user may specify a relocation factor with the SET debug command. This relocation factor is added to every relative address specification that is not overridden by the @C modifier. For example, the command sequence

```
SET RF=517000
DISPLAY 2A0
```

displays the contents of location 5172A0.

The user may display any legal location in his virtual memory by specifying an absolute (virtual) address. To display the contents of the absolute location 516A28, the command is

```
DISPLAY 516A28
```

If possible, the contents of 516A28 are converted according to type and length and printed on the user's terminal. If the address given specifies an illegal, an unreferenced, or a protected page, an error message is printed.

A special form of the SDS parameter, the block parameter, is provided to specify a range of locations. For example, to display the linear array XVAL of dimension 10, the command

```
DISPLAY XVAL(1)...XVAL(10)
```

is used. If the symbol used for the upper-bound address is identical to that of the lower-bound address, it may be omitted, for example,

```
DISPLAY XVAL(1)...(10)
```

Block parameters may be used to specify any range of addresses as shown in the following examples:

```
DISPLAY ALPHA...GAMMA(2)
DISPLAY 9C0...A28@R
DISPLAY 516000...516A28
```

The upper-bound address must be greater than the lower-bound address.

Keyword Modifiers

It is often useful to override the implied or assumed type, length, or scale attributes of an SDS parameter. SDS provides a set of keyword modifiers which temporarily override the attributes of a parameter by explicitly specifying new attributes. A keyword modifier consists of the modifier character "@" followed by a keyword describing the modifier. For instance, the user may wish to display a symbol in its hexadecimal representation. In the following example, the TYPE keyword modifier is applied to the symbol ALPHA. The one character code following the "=" is the type code for hexadecimal formatting, as defined below. The command

```
DISPLAY ALPHA@TYPE=X
```

displays the contents of ALPHA as

```
ALPHA  'X'  0000000A
```

If a length attribute of 8 is also applied, the command is

```
DISPLAY ALPHA@TYPE=X@LENGTH=8
```

and the symbol is displayed as

```
ALPHA  'X'  0000000A  00000014
```

Only the first letter of any keyword modifier need be entered. Hence, the preceding example could be given as

```
DISPLAY ALPHA@T=X@L=8
```

As many keyword modifiers as needed may be appended to a parameter. For block parameters, the TYPE, LENGTH, and SCALE modifiers may be appended to either the upper bound or lower bound address, for example,

```
DISPLAY 516100@T=F@L=4...516120
```

or

```
DISPLAY 516100...516120@T=F@L=4
```

It is often desirable to restrict the use of an SDS parameter to a single control section by the use of the CSECT keyword modifier. This is normally done when a symbol or relative address is defined in more than one control section. For example,

```
DISPLAY BETA@C=SUBR
```

displays the contents of location BETA in the section SUBR.

The symbol BLANK may be used for referring to blank named common sections. For example, the variable A in a blank named common section of a FORTRAN program may be displayed by

```
DISPLAY A@C=BLANK
```

If the program already has a control section or common section by the name of BLANK, the SET command may be used to alter the blank named common symbol. The use of BLANK as a blank named common symbol does not interfere with the use of BLANK for other symbols in the program.

If a control section of a given name appears more than once in the program, the GEN keyword modifier can be used to specify the desired occurrence of that section. The first time a control section of a given name is loaded, it is assigned the generation number 1, the second time it is loaded, it is assigned the generation number 2, and so on. The command

```
DISPLAY BETA@C=SUBR@G=2
```

displays the contents of location BETA in the second occurrence of section SUBR.

The DSECT keyword modifier can be used to restrict an SDS parameter to a particular dummy control section or dsect. If a dsect of a given name appears in more than one assembly, the desired dsect can be specified by appending the CSECT modifier immediately after the DSECT modifier. For example,

```
DISPLAY LOC1@D=DAREA@C=SUBR
```

displays the contents of location LOC1 in the dsect DAREA from the assembly containing the section SUBR. Dynamic or static addressability for the dsect must be provided by the USING command. For example,

```
USING DAREA@C=SUBR GR1
```

causes SDS to use the current contents of GR1 as the base address for DAREA.

For block parameters, the CSECT, GEN, and DSECT parameters apply only to the address (upper bound or lower bound) to which they are appended.

SDS recognizes the following keyword modifiers:

@TYPE=code sets the type attribute to code, where code is any of the single-character type-codes defined below. The default is X.

Code	Type
A	A-type address constant
B	binary
C	character (EBCDIC)
E	floating-point (long or short)
F	fixed-point integer
I	machine instruction
L	logical (FORTRAN)
M	complex (FORTRAN)
P	packed decimal
S	S-type address constant
V	V-type address constant
W	channel command word
X	hexadecimal
Y	Y-type address constant
Z	zoned decimal
#	assembler-generated space (output only)

@LENGTH=i sets the length attribute to i, where i is an unsigned decimal integer. The default is 4.

@SCALE=i sets the binary scale factor to i, where i is a signed or unsigned decimal integer. The default is 0. Binary scale factors follow the conventions used by the 360 assembler language.

@CSECT=cs restricts the parameter to the control section cs.

@GEN=i restricts the parameter to the ith generation of the current control section.

@DSECT=ds restricts the parameter to the dsect ds.

The following modifiers may be used to modify any constituent or keyword modifier in an SDS parameter:

@Q Treat as a symbolic location.

@R Treat as a hexadecimal relative address.

@A Treat as a hexadecimal absolute (virtual) address.

@X Cause hexadecimal conversion.

@D Cause decimal conversion.

The modifier character may be changed by the SET command, so that symbols containing an "@" character can be referred to in SDS parameters.

Predefined Symbols

SDS predefines a small number of "useful" symbols. For instance, to display the contents of all of the general registers, the predefined symbol "GRS" is used, thus,

```
DISPLAY GRS
```

causes the contents of the general registers (0-15) to be displayed. To refer to the ith general register, the symbol "GRi" is used, where "i" is a register number, ranging from 0 - 15, or 0 - 9 and A - F (hex).

Similarly, to refer to the floating-point registers, the symbol "FRS" is used. To refer to the jth floating-point register, the symbol "FRj" is used, where "j" is 0, 2, 4, or 6.

To refer to the current program status word (PSW), the symbol "PSW" is used.

To refer to a program defined location with the same name as an SDS predefined symbol, the @Q modifier must be appended to the symbol. Thus, to display the program defined symbol GR7, the command

```
DISPLAY GR7@Q
```

must be given, whereas, to display general register seven, the command

```
DISPLAY GR7
```

should be used.

Indirection

A level of indirection through any SDS parameter may be achieved by placing the indirection character "$" at the front of the parameter. The parameter is assumed to contain a fullword, byte aligned value which is used by SDS as the effective address. For example, to display the contents of the location whose address is contained in general register six, the command

```
DISPLAY $GR6
```

is used; whereas, to display the contents of general register six, the command

```
DISPLAY GR6
```

is sufficient. As many levels of indirection as needed may be achieved by placing the appropriate number of indirection characters at the front of the parameter. For example, to display the location whose address is contained in the location pointed to by general register two, the command

```
DISPLAY $$GR2
```

is given.

SDS stops at the first illegal address encountered and reports the indirection level. An illegal address is an address which specifies a protected page, an unreferenced page, or a non-existent page. The chain of indirection need not be contained in SDS processed control sections.

The indirection operator has higher precedence than "±" used to add a displacement. For example, the expression $ADDR+4 refers to the location which is 4 bytes past the address which is given in location ADDR. It does not use the contents of ADDR+4 as the location ($).

The indirection character may be changed by the SET command, so that symbols beginning with a "$" can be referred to in SDS parameters.

Input Conversion

Input conversion is used with the MODIFY and SCAN commands. The MODIFY command requires two parameters:

(1) an SDS parameter specifying what locations are to be modified, and
(2) a list of one or more constants delimited with blanks or commas; the entire list is enclosed in primes. The constants are converted according to the type and length attributes of the first parameter.

The SCAN command also requires two parameters initially:

(1) a section name or block parameter specifying a region to be scanned, and
(2) a single constant enclosed in primes. The constant is converted according to the type and length attributes applied to the first parameter.

No modifiers may be appended to the constants themselves. Type, length, and scale modifiers may be appended to the first parameter specified. Conventions for the representation of the different types of SDS constants

are presented below. The TYPE keyword modifier must be applied to the first parameter if the constant type does not agree with the parameter's default type. The term parameter refers to the operand which is being modified or scanned. An error comment is produced if an unsupported type is specified.

A-type adcon - If the length of the parameter is three or four bytes, the constant may be any legal virtual address, relative address or program defined symbol. If the length of the parameter is one or two bytes, the constant is treated as a fixed-point decimal integer of length one or two, respectively. The example below modifies the location ADDR to contain the address of the symbol LOOP.

```
MODIFY ADDR 'LOOP'
```

Binary - Not supported.

Character - Any of the valid 256 punch combinations may be designated in a character constant. Only one character constant may be specified in the second operand of the MODIFY command. Since multiple constants within the second operand are separated by blanks or commas, an attempt to specify two character constants results in interpreting the comma or blank as a character. Special attention must be given to representing primes as characters. Each single prime as a character in the constant must be represented as a pair of primes. The maximum length of a character constant is 256 bytes. Double primes are counted as one character. For the MODIFY command, if the number of characters is less than the length of the parameter, the excess rightmost bytes are filled with blanks. If the number of characters in the constant exceeds the length of the parameter, an error message is given. For the SCAN command, the length of the constant is taken as the actual number of characters specified. In the example below, the location CHAR is modified to contain the character string ABCDE.

```
MODIFY CHAR 'ABCDE'
```

CCW - Not supported.

Complex - A complex constant consists of a pair of floating-point constants, separated by a comma or a blank. The first constant is the real part and the second constant is the imaginary part. If only one constant is specified, only the real part is modified. The length of the entire complex constant is 8 or 16 bytes. Complex constants are used primarily in FORTRAN programs. In the examples below, the location COMP1 is modified to contain 3.0 and 4.0 for the real and imaginary parts of a complex number, respectively, and the location COMP2 is modified

to contain 5.0 as the real part with the imaginary part being unchanged.

```
MODIFY COMP1 '3.0,4.0'
MODIFY COMP2 '5.0'
```

Fixed-point - A fixed-point constant is formed according to the rules for writing floating-point constants. However, unless a binary scale factor is supplied by the parameter being modified, the fractional part is lost. Binary scale factors follow the conventions used by the 360 assembler language, that is, the scale factor multiplies the constant by the specified power of two. A fixed-point parameter may vary in length from one to four bytes. Fixed-point constants are right-justified and padded with leading zeros. In the examples below, the location FIX is modified to the constant 2, and the location FIX2 is modified to the constant 32.

```
MODIFY FIX '2'
MODIFY FIX2@S=3 '4'
```

Floating-point - A floating-point constant is written as a decimal number. As an option, a decimal exponent may follow. The number may be an integer, a fraction or a mixed number. If the length of the parameter is 4 bytes or less, the constant is treated as a single-precision constant; otherwise, it is treated as a double-precision constant. The format of the constant is as follows:

(1) The number is written as a signed or unsigned decimal value. The decimal point can be placed before, within, or after the number. If it is omitted, the number is assumed to be an integer. A positive number is assumed if an unsigned constant is specified.

(2) The exponent is optional. If specified, it is written immediately after the number as En, where "n" is an optionally signed decimal value specifying the exponent of the factor 10. The value of the constant may be in the range of .723700515E+76 to .5397605 35E-78. If an unsigned exponent is specified, a plus sign is assumed. No binary scale factor may be applied.

In the following examples, the parameter XY is modified to 46.415.

```
MODIFY XY '46.415'
MODIFY XY '46415E-3'
MODIFY XY '+.46415E2'
```

Hexadecimal - A hexadecimal constant consists of one or more of the hexadecimal digits 0-9 and A-F. Only one hexadecimal constant may be specified in the second operand. The maximum length of a hexadecimal constant is 256 bytes. Constants that contain an even number of digits are translated as one byte per pair of digits. If an odd number of digits is specified, the leftmost byte has its leftmost four bits filled with zeros, while the rightmost four bits contain the first digit. Hexadecimal constants are left-justified. There is no padding. In the examples below, the locations HEX1 and CHAR are modified to the hexadecimal constant C1C2C3C4C5.

```
MODIFY HEX1 'C1C2C3C4C5'
MODIFY CHAR@T=X 'C1C2C3C4C5'
```

Instruction - An "instruction" constant consists of a 360 assembler mnemonic and a hexadecimal operand of the appropriate length. The operand is separated from the mnemonic by one or more blanks. No extended mnemonics (such as BNM) may be used. The length of the instruction constant need not agree with the length of the instruction being modified. From the sample program above, the location AD is modified to the instruction "S 1,C".

```
MODIFY AD 'S 10F014'
```

Logical - Logical constants are ".TRUE." and ".FALSE.", which are converted internally to the fixed-point numbers 1 and 0, respectively. If neither ".TRUE." nor ".FALSE." is specified, the constant is treated as a character string. Logical constants are used primarily in FORTRAN programs. In the examples below, the location LOG1 is modified to the constant ".TRUE." and the location LOG2 is modified to the character A.

```
MODIFY LOG1 '.TRUE.'
MODIFY LOG2 'A'
```

Packed & zoned - A packed or zoned decimal constant is written as a signed or unsigned decimal value. If the sign is omitted, a plus sign is assumed. The existence of a decimal point in no way affects the conversion of a decimal constant. In effect, the decimal point is ignored. Scaling and exponent modifiers may not be used with decimal constants. The maximum length of a decimal constant is 16 bytes. If zoned decimal conversion is being performed, each decimal digit is translated into one byte. The rightmost byte contains the sign as well as the rightmost digit. For packed decimal conversion, each pair of decimal digits is packed into one byte. The rightmost digit and the sign are translated into the rightmost byte. If an even number of packed decimal

digits is specified, the leftmost four bits in the leftmost byte are set to zero and the rightmost four bits contain the first digit. If the constant requires fewer bytes than the length of the parameter, the bits of each added byte are set to zero for packed decimal conversion. For zoned conversion, the decimal digit zero is placed in each added byte. If the constant requires more bytes than the parameter specifies, no modification occurs.

```
MODIFY PACK1 '23'
MODIFY ZONE1 '-44'
```

S-type adcon - Not supported.

V-type adcon - V-type adcons are treated the same as A-type adcons. In the example below, the location VCON is modified to the virtual address 512600.

```
MODIFY VCON '512600'
```

Y-type adcon - Y-type adcons are treated as fixed-point decimal constants whose length depends on the length of the parameter being modified. In the example below, the location YCON is modified to the integer constant 2.

```
MODIFY YCON '2'
```

Output Conversion

All of the conversion types listed with the description of the TYPE keyword modifier are supported by the SDS output conversion routines.

The following conventions are used when SDS attempts to display illegal data:

Character - If a hexadecimal code is encountered which has no character equivalent, a question mark is substituted.

Floating-point - If the length of the number exceeds 8 bytes, the number is printed in hexadecimal format.

Fixed-point - If the length of the number exceeds 16 bytes, the number is printed in hexadecimal format.

Instruction - If the opcode field of an instruction does not correspond to a machine operation, the instruction is printed as two hex digits surrounded by asterisks. The instruction is assumed to be 2 bytes in length (RR type).

Packed & zoned - If the length of the number exceeds 16 bytes, the number is printed in hexadecimal format.

Current Symbol Character

The current symbol character "*" can be used to represent the last SDS parameter specified in a debug command. This is most useful in SDS command sequences such as

```
DISPLAY ABC
MODIFY * '1'
```

which displays the contents of location ABC and then modifies that location to the constant '1'.

The current symbol character may have a displacement and/or modifiers appended to it, but not an index. For example,

```
DISPLAY ABC *@T=X@L=8
```

displays the contents of location ABC according to its type and length, and then displays ABC in hexadecimal format with a length of 8 bytes.

```
DISPLAY ABC(10) *+4
```

displays the locations ABC(10) and ABC(10)+4.

For the case of block parameters, the current symbol is the lower-bound symbol. For example,

```
DISPLAY ALPHA...BETA *@T=X
```

causes the range from ALPHA to BETA, to be displayed, after which the contents of ALPHA in hexadecimal format, are also displayed.

Note that many debug commands refer to SDS parameters and, therefore, change the value of the current symbol character. Hence, the user should be aware of the value that the current symbol character is representing.

PROGRAM RETURNS, DYNAMIC LOADING, AND INTERRUPT PROCESSING

Whenever the user's program returns to the system by calling subroutines SYSTEM, MTS, MTSCMD, or ERROR, SDS intercepts the return and returns control to debug command mode. For a return to the system or a call to SYSTEM, the program may not be continued with the CONTINUE command; for the other subroutine calls, the program may be continued. For a call to MTSCMD, the command specified will have been executed.

Whenever the user's program dynamically loads another section via a call to the subroutines LINK, LOAD, or XCTL, SDS intercepts the call and returns control to debug command mode. The sections specified in the subroutine call have been loaded at this point, but not entered. The user may then set breakpoints, if he desires, before continuing the program. The interception of calls to LINK, LOAD, or XCTL may be enabled by the debug command

```
SET XFR=ON
```

Whenever a program interrupt, an attention interrupt, a timer interrupt, or an I/O error occurs during the execution of the program, SDS normally intercepts the error condition and returns control to debug command mode.

In the case of a program interrupt, the location of the instruction causing the interrupt is printed. For an imprecise interrupt (protection exception), an indication that the interrupt was imprecise is also printed. The user's program may be restarted by the CONTINUE command, unless the interrupt was imprecise, in which case, the GOTO or RUN command must be used.

In the case of an attention interrupt, the location at which the interrupt occurred is printed. The user's program may be restarted by the CONTINUE command.

In the event that the user's program has called the system subroutines PGNTTRP or ATTNTRP, SDS may or may not regain control depending on what the user's interrupt routines do. However, the user may disable his own interrupt routines by entering the debug command

```
SET PGNT=OFF ATTN=OFF
```

When the interrupt is taken, control is returned immediately to SDS. If the user restarts his program with the CONTINUE command, execution is restarted at the instruction at which the interrupt was taken. The user's interrupt routines are not called. Normal interrupt processing may be resumed by setting the PGNT or ATTN options ON. Note that all FORTRAN programs automatically call PGNTTRP to allow the FORTRAN library to process program interrupts. For SDS to process program interrupts in a FORTRAN program, PGNT must be set OFF.

In the case of a timer interrupt, the location at which the interrupt occurred is printed. The exceeding of a local time limit is the only type of timer interrupt which causes control to be returned to debug mode unless the exit routine returns to the system.

In the case of an I/O interrupt, a system error message (if any) is printed. If the user has suppressed the intercept of the I/O error either by calling the subroutines SETIOERR or SIOERR or by specifying the ERRRTN I/O modifier on the I/O call, control is not returned to debug mode.

If the user's program causes an MTS program interrupt (often caused by passing an illegal parameter to an I/O subroutine), control is returned to debug mode with the message that execution has been prematurely interrupted.

Whenever a program interrupt, an attention interrupt, an I/O interrupt, a timer interrupt, or an intercepted call to LINK, LOAD, or XCTL occurs, SDS changes its input source to read from *MSOURCE* and its output sink to write on *MSINK*.

BREAKPOINT PROCESSING

SDS provides the user with the facility of setting several different types of breakpoints in strategic locations in his program. When encountered during program execution, these breakpoints cause SDS to assume control of the program and take a certain course of action. Each breakpoint inserted in the user's program is a halfword code which replaces the contents of the location at which it is inserted and which causes a special program interrupt and returns control to SDS. The halfword portion of the instruction that is replaced by the breakpoint code is saved for later execution when the program resumes execution. The different types of breakpoints that are currently available are described below.

Global Breakpoints

A global breakpoint is used to return control to debug command mode. The user may then enter any debug ccmmand. Global breakpoints are set by the BREAK command. For example,

```
BREAK AD
```

sets a simple breakpoint at location AD. For further details of global breakpoint processing, see the description of the BREAK command below.

Local Breakpoints

A local breakpoint is used to return control to debug command mode. Local breakpoints are set by the parameters to the RUN and CONTINUE commands, or the second and successive parameters to the GOTO command. For example,

```
RUN AD
```

sets a local breakpoint at location AD and then initiates execution of the user's program at the entry point. If more than one local breakpoint is set, the first one encountered returns control to debug command mode. Local breakpoints are in effect only for the duration of the current command that set them; they are automatically erased before the user enters his next command.

At-points

An at-point is used to cause SDS to process a list of prestored commands. This list of prestored commands is set up by the AT command. (See the description of the AT command.)

Breakpoints may not be set at an instruction which is the object of an execute instruction, at an instruction which is modified by the program, or at an instruction preceding a BPI instruction. For execute instructions, the breakpoint should be set at the execute instruction itself. If the location specified is not halfword-aligned, the breakpoint is not set. A warning is issued if the user sets a breakpoint in a data item.

THE SDS SIMULATOR

A program simulator is provided by SDS to allow the user to step through his program one or more instructions at a time. All instructions that are stepped through are simulated by SDS instead of executing normally.

The simulator is invoked by the STEP command specifying the number of instructions to be stepped, for example,

```
STEP 3
```

causes the next three instructions to be stepped. If no step count is given, a step of 1 is assumed. Stepping starts with the current address contained in the PSW. Upon completion of the stepping, control is returned to debug command mode.

If an abnormal condition occurs during stepping, the stepping is terminated at the current instruction and control returns to debug command mode. If the program branches to a legal low-core symbol such as SCARDS or SPRINT, the routine is executed instead of being simulated and stepping resumes upon the return from that routine. The execution of the routine does not count as a step. If the program branches to an address less than 300000 which does not correspond to a legal low-core symbol, stepping is terminated and a warning message is printed. The user may restart the program with either a CONTINUE or a GOTO command.

If a program interrupt or an attention interrupt occurs and the user has specified an exit routine via the PGNTTRP or ATTNTRP subroutines, these routines are also simulated until the specified stepping count is exhausted. This may be overridden by the PGNT and ATTN options of the SET command.

FORTRAN users should note that the STEP command specifies machine language instructions in the count. To step a specified number of FORTRAN instructions, the CONTINUE command should be used specifying temporary breakpoints.

CONTROL SECTION PROCESSING

When a program of one or more modules[1] is initially loaded by SDS, the control sections, common sections, and dummy control sections of the modules are entered into the SDS map in the order in which they appear in the load module(s). These modules comprise a single map control block within the SDS map. If, during execution, the program calls the loader to load more modules, these also are processed by SDS and entered into the SDS map in a new map control block.

A separate map control block is created for each call to the loader. Hence, the sections that comprise each map control block have a unique storage index number.

The first time a control section name appears in the SDS map, it is assigned a generation number of 1. If a control section is loaded which has the same name as a previously processed control section in an earlier map control block, that section is assigned a new generation number one greater than the last previous generation number used for a section by that name; hence, for each subsequent occurrence of a control section for a given name, there is assigned to it a unique generation number. This generation number can be used via the GEN keyword modifier to distinguish between the different occurrences of control sections with the same name. Dummy control sections (dsects) do not have generation numbers assigned to them.

Initially, the first module in the SDS map becomes the current module and the remaining modules are globally opened for referencing. If SDS cannot locate a given symbol in the current module, the remaining modules are searched. If the symbol is located, SDS searches through all remaining modules for a match to determine whether or not the symbol is multiply defined. A warning message is printed for a multiply defined symbol and the first definition is used. If a relative address can be located in the current module, no further searching occurs.

The order of searching through the SDS map for a symbol is as follows:

(1) If the symbol is a predefined symbol, the search terminates immediately.
(2) If only one module of the map is open (see below), only that module is searched. The search covers all control sections and defined dsects of that module.
(3) If the entire map is open, the most recent generation of the first section is searched. If the symbol is not found, the next oldest generation is searched. If the generation string does not contain the symbol, the most recent generation of the next section is searched. If no match is found in any generations of sections loaded in the first map control block, the sections in the next map

---

[1] A module is defined as a sequence of load records up to and including an END record. For FORTRAN programs, each main program and subroutine produces a separate object module.

control block are searched according to the same algorithm. No section is searched more than once. If a match is found in any section, the remainder of the map is searched for a duplicate symbol. If none is found, the search terminates; otherwise a warning is printed and the first match is used. If a match is found in an undefined dsect, the search continues looking for a match in a defined section. If such a match is found, and no futher duplication exists, the search is terminated with no warning message.

The CSECT command provides a means of specifying a new current module. The module containing the section named by the command becomes the new current module and the remaining modules are closed; that is, checking for multiply defined symbols is suppressed and, if a symbol cannot be located in the current module, no further searching takes place. If the section specified is a control section or dsect, all control sections and dsects in that module are open for searching; if the section specified is a common section, only that common section is open for searching. For example, the command

```
CSECT SUBR
```

specifies the module containing the section SUBR as the current module. If the CSECT command specifies a common section, only that common section is taken as the current module; all other symbols that may be in the same module are ignored. If the command

```
CSECT i
```

is entered, where i is an unsigned decimal integer, the module containing the ith blank-named control section in the SDS map becomes the current module. This is the only way to refer to blank-named (private) control sections in the map. A blank-named common section may be referred to by the blank-name common symbol BLANK.

If the command

```
CSECT *
```

is entered, the first module in the SDS map again becomes the current module and the remaining modules are opened. The open-map character "*" may be changed by the SET command.

Object modules composed of multiple assemblies may have name conflicts between control sections, common sections and dsects. SDS observes the following conventions when such conflicts occur within a single map control block:

Situation	Action Taken
A CSECT has the same name as a previously defined CSECT.	The original CSECT is retained and the duplicate CSECT is ignored, unless it is a blank-named CSECT.

A CSECT has the same name as a previously defined COMMON section.	The symbols of the CSECT are merged with the symbols of the COMMON section.
A CSECT has the same name as a previously defined DSECT.	Both the CSECT and the DSECT are retained.
A COMMON section has the same name as a previously defined CSECT.	The CSECT is marked as a COMMON section in the SDS map, and the symbols in the COMMON section are merged with the symbols of the CSECT.
A COMMON section has the same name as a previously defined COMMON section.	The original COMMON section is retained and the symbols of each are merged.
A COMMON section has the same name as a previously defined DSECT.	Both the DSECT and the COMMON section are retained.
A DSECT has the same name as a previously defined CSECT.	Both the DSECT and the CSECT are retained.
A DSECT has the same name as a previously defined COMMON section.	Both the DSECT and the COMMON section are retained.
A DSECT has the same name as a previously defined DSECT in the same module.	The original DSECT is retained and the symbols of each are merged.
A DSECT has the same name as a previously defined DSECT in a different module.	Both DSECTs are retained.

MISCELLANEOUS CONCEPTS

Terse Mode

SDS provides a terse mode of operation which eliminates or shortens many of the confirmation and diagnostic messages printed by SDS. This mode is activated by the command

```
SET TERSE=ON
```

and is deactivated by the command

```
SET TERSE=OFF
```

The following messages are eliminated in terse mode:

(1) READY. This message is often printed when SDS is ready to accept a command.
(2) DONE. This message is often printed after SDS has taken some action such as setting or restoring a breakpoint.
(3) Verification by the MODIFY command. The MODIFY command normally verifies the modification by printing both the old and new values of the location being modified.

The following messages are shortened in terse mode:

(1) The breakpoint and at-point interrupt messages give only the address of interruption and do not identify the type of interruption.
(2) The call to SYSTEM, ERROR, MTS, and MTSCMD messages do not give the GR14 return address.

Automatic Error Dumping in Batch

An automatic error dumping facility similar to that provided by the MTS $ERRORDUMP command is provided for batch users. In the event of an error condition occurring during the execution of the program, a symbolic dump of the program is given. This dump includes the PSW, the general and floating-point registers, and all of the data storage locations in the program. Instructions and other areas not covered by the symbol table are excluded. This facility may be activated by the command sequence

```
$SET DEBUG=ON
$SDS SET ERRORDUMP=ON
$RUN FDname
```

where FDname is the file containing the user program to be executed. Note that the MTS $RUN command has been given instead of the $DEBUG command. The error-dump facility may be deactivated by the command

```
$SET DEBUG=OFF
```

or

```
$SDS SET ERRORDUMP=OFF
```

Terminal users may obtain a symbolic dump by the DUMP debug command. A sample of the symbolic dump format is given in the section "Introduction to Debug Mode for FORTRAN".

Using SDS Without a Loaded Program

Several of the debug commands may be used successfully even if SDS has not processed the loaded program or if there is no currently loaded program. For example, the user may use the output conversion facilities of SDS to display selected locations of virtual memory, for example, the debug command

```
DISPLAY 516260@T=C@L=32
```

displays in character format the 32 bytes starting at location 516260, if it is a valid virtual address. The input conversion facilities may be used to modify selected storage. For example, the debug command

```
MODIFY 516260@T=C@L=8 'ABCDABCD'
```

causes the 8 bytes starting at location 516260 to be modified to the character string ABCDABCD.

The above command may be given from MTS command mode in the form of a "one-shot" $SDS command:

```
$SDS MODIFY 516260@T=C@L=8 'ABCDABCD'
```

After the one-shot command is executed, control is returned to MTS command mode.

Initializing, Resetting, and Terminating SDS Processing

When SDS is initialized, an area of system storage is assigned to SDS and all of the default SDS options are set. SDS is initialized under any of the following conditions:

(1) If DEBUG=OFF is specified, (the default), SDS is initialized with the first $DEBUG or $SDS command in the job or after SDS has been terminated (see below).
(2) If $SET DEBUG=ON is specified, SDS is initialized with the first $DEBUG, $RUN, $LOAD, $RESTART, $START, or $SDS command in the job or after SDS has been terminated.

When SDS is reset, all of the loaded program symbol table information (if any) is released and certain SDS tables are released. The basic SDS work storage remains and the SDS SET options remain in effect (except for the INPUT, OUTPUT, ENTRY, PAR, and logical I/O unit assignments). Input commands are read from *SOURCE* and output is written on *SINK*. SDS is reset under any of the following conditions:

(1) During initialization (see above).
(2) With each $DEBUG, $RUN, and $LOAD command if $SET DEBUG=ON is specified.
(3) With each $DEBUG command if $SET DEBUG=OFF is specified.

When SDS is terminated, all of the loaded program symbol table information (if any) is released and the basic SDS work storage is released. SDS is terminated under any of the following conditions:

(1) The STOP debug command is given.
(2) The $SET DEBUG=OFF command is given.
(3) The $UNLOAD CLS=SDS command is given.
(4) A $RUN or $LOAD command is given (if $SET DEBUG=OFF is specified).

If a program interrupt occurs in SDS and SDS is not in the process of modifying any of its symbol table information storage areas, SDS returns to debug command mode and prompts the user for permission to continue. If the user replies affirmatively, another debug command may be given; otherwise, SDS error returns to MTS command mode. If SDS is in the process of modifying the symbol table, an SDS error return to MTS occurs immediately.

DEBUG COMMAND DEFINITIONS

On the following page is a list of the debug commands available for SDS. Parameters for each command may be separated by commas or blanks. Some of the parameter terms used are:

(1) simple parameters -- these are parameters of the form S(i)±j.

(2) parameters -- these are either simple parameters or parameters of the form S(i)±j...S(i)±j.

The following notation conventions are used in the prototypes of the commands:

lower case - represents a generic type which is to be replaced by an item supplied by the user.
upper case - indicates material to be repeated verbatim in the command.
brackets [ ] - indicate that material within the brackets is optional.
braces {} - indicate that the material within the braces represents choices, from which exactly one must be selected. The choices are separated by vertical bars.
dots ... - indicate that the preceding syntactic unit(s) may be repeated.
underlining - indicates the minimum abbreviated form of the command or parameter. Longer abbreviations are accepted.

Note that for descriptions of the ATTNTRP, INPUT, LENGTH, OUTPUT, PGNTTRP, PREFIX, RF, SCALE, TERSE, and TYPE commands, see the appropriate SET command options. These commands are implemented as individual commands, but are not individually documented.

Summary of Debug Command Prototypes

Commands	Parameters
ALTER	simple-parameter {'value ...'|simple-parameter}
AT	simple-parameter ...
ATTNTRP	{ON|OFF}
ATTRIBUTE	simple-parameter ...
BREAK	simple-parameter ...
CLEAN	[BREAKPOINTS][AT-POINTS]
COMMENT	[text]
CONTINUE	[simple-parameter ...]
CSECT	{section|i|*}
DISPLAY	parameter ...
DROP	{dsect-name|simple-parameter} ...
DUMP	[{PSW|GRS|FRS|section} ...]
END	
GOTO	simple-parameter [simple-parameter ...]
HEXDISPLAY	parameter ...
IGNORE	breakpoint-label {i|simple-parameter}
INPUT	FDname
LENGTH	i
LIST	[BREAKPOINTS][AT-POINTS]
MAP	[{FULL|DSECT}]
MODIFY	simple-parameter {'value ...'|simple-parameter}
MTS	[command]
OUTPUT	FDname
PGNTTRP	{ON|OFF}
PREFIX	character
QUALIFY	simple-parameter ...
RESET	
RESTORE	[simple-parameter ...]
RF	hexadecimal-constant
RUN	[simple-parameter ...]
SCALE	i
SCAN	[{section|address1...address2|*} 'value']
SDS	
SET	keyword-parameter ...
STEP	[i]
STOP	
SYMBOL	simple-parameter ...
TERSE	{ON|OFF}
TYPE	code
USING	dsect-name address

ALTER

DEBUG COMMAND DESCRIPTION

Prototype: ALTER simple-parameter {'value ...'|simple-parameter}

Action: The ALTER command is identical to the MODIFY command. See the description of the MODIFY command.

April 1974

AT

DEBUG COMMAND DESCRIPTION

Prototype: AT simple-parameter ...

Action:

(1) An at-point is inserted at each location specified in the parameter list. The at-point code is an X'0002' which replaces the instruction at the location specified; the instruction replaced is saved for later execution when execution of the program is resumed.

(2) SDS enters into at-insertion mode (indicated by the at-prefix character "%"). All the commands entered during at-insertion mode are saved in a command list for later processing. If the user enters a null command line during at-insertion mode, the previously-entered command is deleted from the list of saved commands. When an end-of-file or END command is entered, SDS resumes normal command processing. If another AT command is entered during at-insertion mode, the first command list is terminated and a second command list is started.

When an at-point is encountered during the execution of the user's program, each of the commands in the command list for that at-point is executed in sequence. When the last deferred command has been executed, control returns to the user's program which resumes normal execution. If control is to be returned to debug command mode rather than to the program, the SDS command should appear immediately before the END command since the SDS command terminates the processing of the command list. An abnormal condition also terminates command list processing.

Comments:

At-points are automatically announced if the command list causes an output message to be generated (for example, the output from the DISPLAY command), or if there is no command list for the at-point. The COMMENT commmand may be used to cause comments or announcements of at-points when no output messages would otherwise be generated.

An at-point may not be set at an instruction which is the object of an execute instruction, at an instruction which is modified by the program, or at an instruction preceding a BPI instruction. For an execute instruction, the at-point should be set at the execute instruction itself. If the location specified is not halfword-aligned, the at-point is not set. A warning is issued if the user sets an at-point in a data item.

At-points may be removed by either the RESTORE or the CLEAN commands.

Examples:

```
AT LOOP
DISPLAY XVAL YVAL N
DISPLAY GR1 FR0
END
```

In the first example, each time location LOOP is reached, the current values of XVAL, YVAL, N, general register 1 and floating-point register 0 are displayed.

```
AT E00@R 504056
DISPLAY GRS
AT LOC2
MODIFY FZ '0000'
DISPLAY Y(1)...(6)
SDS
END
```

In the second example, each time the relative location E00 or the absolute location 504056 is reached, the general registers are displayed. Each time location LOC2 is reached, location FZ is zeroed and the first six elements of the Y array are displayed; control then returns to debug command mode rather than to the program.

```
AT IS#5
DISPLAY A(1,1)...(1,10)
CONTINUE IS#8
DISPLAY A(1,1)...(1,10)
END
```

In the third example, the first ten elements of the FORTRAN array A are displayed at location IS#5 and execution is resumed by the CONTINUE command specifying a local breakpoint at IS#8. When the local breakpoint is encountered at IS#8, the array is displayed again; execution is then resumed since the at-point command list processing is concluded.

April 1974

ATTRIBUTE

DEBUG COMMAND DESCRIPTION

Prototype: ATTRIBUTE simple-parameter ...

Action: The attributes for each parameter in the list are displayed; these include:

the loaded absolute address,
the relative address,
the section name,
the section generation number (if different from one),
the type,
the length,
the duplication factor (if different from one),
the scale factor (if different from zero), and
the dimension number (if a FORTRAN array).

The one-character codes for the type are defined with the description of the TYPE keyword modifier.

Example:

```
ATTRIBUTE GAMMA
```

The attributes of GAMMA are displayed as

```
GAMMA:
LA=514020 RA=000020 SECTION=PROG TYPE=F LEN=4 DUP=20
```

BREAK

DEBUG COMMAND DESCRIPTION

Prototype: BREAK simple-parameter ...

Action: A global breakpoint is inserted at each location specified in the parameter list. The breakpoint code is an X'0001' which replaces the instruction at the location specified; the instruction replaced is saved for later execution when execution of the program is resumed.

When the program being debugged attempts to execute the instruction at the breakpoint, control is returned to SDS which announces the location of the breakpoint and prompts for its next command. The instruction at the breakpoint has not yet been executed. The status of the program is preserved and may be examined and modified with the appropriate commands. The program may be restarted with the CONTINUE or STEP commands which execute the instruction at the breakpoint and resume normal sequencing. To restart at some other point, the GOTO command may be used.

Breakpoints may be removed with either the RESTORE or the CLEAN commands.

A breakpoint may not be set at an instruction which is the object of an execute instruction, at an instruction which is modified by the program, or at an instruction preceding a BPI instruction. For an execute instruction, the breakpoint should be set at the execute instruction itself. If the location specified is not halfword-aligned, the breakpoint is not set. A warning is issued if the user sets a breakpoint in a data item.

For FORTRAN programs, the user may set breakpoints by specifying the external (user-defined) or internal (source listing) statement numbers. The "#" must be used to prefix the external statement number, for example,

```
BREAK #10
```

and "IS#" must be used to prefix the internal statement number, for example,

```
BREAK IS#10
```

Note that only those statement numbers which define executable FORTRAN statements may be used. All others, such as DIMENSION, DATA, COMMON, SUBROUTINE, FUNCTION, ENTRY, EQUIVALENCE, or FORMAT statements are undefined.

Example:

```
BREAK  LOOP 206 513100
```

Global breakpoints are set at the symbolic location LOOP, the relative location 206, and the absolute location 503100.

CLEAN

DEBUG COMMAND DESCRIPTION

Prototype: CLEAN [BREAKPOINTS][AT-POINTS]

Action: If no parameter is specified, all breakpoints set by the BREAK command and all at-points set by the AT command are deleted. If a parameter is given, then only the specified type is deleted.

Example:

```
CLEAN A
```

All at-points are removed from the program.

April 1974

COMMENT

DEBUG COMMAND DESCRIPTION

Prototype: COMMENT [text]

Action: If the user has entered the command in at-insertion mode, the comment is printed when the at-point command list is processed.

If the user is in debug command mode, no action is taken.

Example:

```
AT LOOP
COMMENT SKIP 2 INSTRUCTIONS AT LOOP
GOTO LOOP(3)
END
```

At the location LOOP, the comment "SKIP 2 INSTRUCTIONS AT LOOP" is printed; then, execution resumes at location LOOP(3).

CONTINUE

DEBUG COMMAND DESCRIPTION

Prototype: CONTINUE [simple-parameter ...]

Action: Execution of the program is resumed from the point of the last interrupt. If a breakpoint was encountered, execution begins with the instruction at the breakpoint. If an attention interrupt or program interrupt had been taken, execution begins at the location specified by the PSW. In the event of an imprecise program interrupt, a GOTO or a RUN command must be used to restart the user's program.

If a simple parameter is specified, a local breakpoint is set at the location specified. This breakpoint code is an X'0004' which replaces the instruction. When the program encounters a local breakpoint, control is returned to debug command mode. Local breakpoints are in effect only for the duration of the command, and are automatically erased before the user enters his next command.

The CONTINUE command may be used to initiate execution of the program if the initial values of the registers and/or PSW have been modified.

Example:

```
CONTINUE LOC2
```

A local breakpoint is set at location LOC2 and execution of the program is resumed.

CSECT

DEBUG COMMAND DESCRIPTION

Prototype: CSECT {section|i|*}

Action: If section is given, the module containing the section named becomes the new current module and the remaining modules are closed; that is, checking for multiply defined symbols is suppressed, and, if a symbol cannot be located in the current module no further searching takes place. If section specifies a control section or dsect, all control sections and defined dsects in that module are open for searching; if section specifies a common section, only that common section is open for searching.

If i is given, the module containing the ith blank-named control section loaded becomes the current module and the remaining modules are closed (see "Control Section Processing" above).

If the open-map character "*" is given, the first module loaded again becomes the current module and the remaining modules are opened.

Comment: To specify a module containing a dsect as the current module, section must be specified as

dsect@C=csect

where "csect" is the name of the control section with which the dsect was assembled. The dsect must be previously defined with a USING command.

Examples:

```
CSECT PROG
```

The module containing the section PROG becomes the current module.

```
CSECT DAREA@C=SUBR
```

The module containing the dsect DAREA in the section SUBR becomes the current module.

DISPLAY

DEBUG COMMAND DESCRIPTION

Prototype: DISPLAY parameter ...

Action: Each simple or block parameter (or element in the range of a block parameter) is converted according to its type and length and printed along with a one character code which indicates the parameter's type. The type codes are defined with the description of the TYPE keyword modifer.

If the type and length attributes of a parameter are unknown, or if a parameter specifies an address which is incorrectly aligned with respect to its type, the contents of the byte specified by the parameter, and the contents of the next three bytes, are printed in hexadecimal format.

To display a large block of storage, use the DUMP or HEXDISPLAY commands if possible for economy.

Example:

```
DISPLAY ALPHA
```

The location ALPHA is displayed according to its type and length attributes.

April 1974

DROP

DEBUG COMMAND DESCRIPTION

Prototype: DROP {dsect-name|simple-parameter} ...

Action: If dsect-name is specified, that dsect loses its addressability. (See the description of the USING command.)

If simple-parameter specifies a general register, all dsects covered by that register are no longer addressable by SDS (that is, symbols and relative addresses within that dsect cannot be accessed). If simple-parameter specifies a location, all dsects based at the virtual address corresponding to that location lose their addressability.

If the parameter may be interpreted both as a dsect name and as a symbol, it is assumed to be a dsect name.

Example:

```
DROP GR1
```

The dsects covered by GR1 are released and are no longer addressable.

DUMP

DEBUG COMMAND DESCRIPTION

Prototype: DUMP [{PSW|GRS|FRS|section} ...]

Action: A symbolic dump of the sections specified is given. The dump includes only the variable storage of the section; instructions and areas not covered by the symbol table are excluded.

For each item dumped, the relative address, the symbolic name, the data type, the converted value, and the hex value (if the data type is not X) is dumped.

If PSW is specified, the hex and symbolic address forms are given.

If GRS is specified, the general registers are dumped in hex, fixed-point decimal, and symbolic address format.

If FRS is specified, the floating-point registers are dumped in hex and floating-point decimal format.

If no parameter is specified, the PSW, the registers, and all sections are dumped.

Comment: Since the output from this command may be extensive, terminal users should set the output device to a file or *PRINT* via the SET OUTPUT=FDname command.

A sample of the symbolic dump format is given in the section "Introduction to Debug Mode for FORTRAN".

For batch users, an automatic symbolic dump may be obtained in the event of an abnormal program termination via the SET ERRORDUMP=ON debug command. This facility is similar to the $ERRORDUMP command in MTS command mode which produces a hexadecimal dump of a program in the event of an abnormal program termination.

Example:

```
SET OUTPUT=*PRINT*
DUMP
```

A symbolic dump of the PSW, the registers, and all loaded sections are produced on *PRINT*.

April 1974

END

DEBUG COMMAND DESCRIPTION

Prototype: END

Action: If the user is in at-insertion mode, the current sequence of commands being entered into command list associated with the at-point is terminated, and control is returned to debug command mode.

If the user is in debug command mode, no action is taken.

Example:

```
AT LOOP
DISPLAY GR1
END
```

The command list for the at-point at location LOOP is terminated by the END command and control is returned to debug command mode.

GOTO

DEBUG COMMAND DESCRIPTION

Prototype: GOTO simple-parameter [simple-parameter ...]

Action: Execution of the program is resumed at the location specified by simple-parameter. The parameter must specify a halfword-aligned address.

If a second simple-parameter is specified, a local breakpoint is set at the location specified. This breakpoint code is an X'0004' which replaces the instruction. When the program encounters a local breakpoint, control is returned to debug command mode. Local breakpoints are in effect only for the duration of the command, and are automatically erased before the user enters his next command.

The GOTO command may be used to initiate the execution of the program if a different entry point is desired.

Example:

```
GOTO LOC1 LCC2
```

A local breakpoint is set at location LOC2 and execution of the program is resumed at location LOC1.

April 1974

HEXDISPLAY

DEBUG COMMAND DESCRIPTION

Prototype: HEXDISPLAY parameter ...

Action: Each simple or block parameter in the list is dumped in the hexadecimal format used by the SDUMP subroutine. (See the description of the SDUMP subroutine in Volume 3.)

Example:

```
HEXDISPLAY 5002A0...5002EC
```

The 96 bytes starting at location 5002A0 are dumped in hexadecimal format.

IGNORE

DEBUG COMMAND DESCRIPTION

Prototype: IGNORE breakpoint-label {i|simple-parameter}

Action: Ordinarily, when control reaches a global breakpoint or an at-point in the program, execution is interrupted and control returns to SDS. The IGNORE command provides a means of suppressing this interruption each time control reaches the breakpoint, for a total of i times, where i is either a decimal integer or the contents of the location specified by a simple parameter. The i+1st time control reaches the breakpoint, the interruption is taken as usual.

If a simple parameter is specified, the parameter's type may be fixed-point (fullword or halfword), floating-point (long or short), or hexadecimal (4 bytes or less). A simple parameter that is in the form of a relative address must have the @R modifier appended.

The ignore count may not be greater than 65535.

Example:

```
IGNORE  LOOP 10
```

The breakpoint at the location LOOP is ignored 10 times.

April 1974

LIST

DEBUG COMMAND DESCRIPTION

Prototype: LIST [BREAKPOINTS][AT-POINTS]

Action: If BREAKPOINTS is specified, a list of the currently set breakpoints created by the BREAK command is produced. If AT-POINTS is specified, a list of the currently set at-points created by the AT command is produced. If neither BREAKPOINTS nor AT-POINTS is specified, all currently set breakpoints and at-points are listed. In all cases, the most recently set breakpoints or at-points are listed first.

Example:

```
LIST A
```

A listing of all the currently set at-points is produced.

MAP

DEBUG COMMAND DESCRIPTION

Prototype: MAP [{FULL|DSECT}]

Action: A loader-like map is produced, listing each control section and common section in the user's program. The map includes the section name, section type, the section length, the loaded address, the relocation factor, and the storage index number. If the storage index number is omitted, then it is the same as the storage index number of the previous section.

Blank-named (private) control sections are specified by unsigned decimal integers assigned according to their order in the SDS map. This integer is the only way to refer to a blank-named control section. A blank-named common section is specified in the map as a blank symbol. The blank-name common symbol (initially BLANK) is used to refer to a blank-named common section.

If the FULL parameter is specified, the map includes all dsects and library loaded sections. If the dsect is undefined, the address field is blank; if the dsect is defined, the address field contains the current address definition for that dsect. Addresses 000001 through 00000F are used to indicate that the dsect is defined by the current contents of GR1 through GRF (GR15), respectively.

If the DSECT parameter is specified, the map only includes the currently defined dsects; control sections, common sections, and undefined dsects are omitted.

The symbols used for the map type are:

SD	control section definition
CM	common section definition
DS	dsect definition
LSD	library control section definition
LCM	library common section defintion
LDS	library dsect definition

Example:

```
MAP FULL
```

A full map is printed as follows:

```
NAME      TYPE  LENGTH ADDRESS   RELOC   SI#
MAIN        SD  000268  5034C8  5034C8  0080
COM1        CM  000070  503730  503730
DSEC1       DS  0000F4
```

```
COM2        CM   000020   5037A0   5037A0
SUBR        SD   000228   5037C0   5037C0
DSEC2       DS   00002C   000009
SQRT       LSD   000032   5039F0   5039F0
```

MODIFY

DEBUG COMMAND DESCRIPTION

Prototype: MODIFY simple-parameter {'value ...'| simple-parameter}

Action: The first parameter specifies the locations that are to be modified and the second parameter specifies the values to be used for the modification.

value ... specifies a list of one or more constants delimited by blanks or commas. The entire list is enclosed in primes. The constant or list of constants is placed in the location specified by the parameter. The constants are converted according to the type and length attributes of the first parameter. No modifiers may be appended to the constants themselves.

If the second parameter specifies a location, then the hexadecimal contents of that location is used for the modification. The length used is the length attribute of the first parameter.

Comments: The use of type, length and scale affects only the first operand in cases where a list is used.

When the modifier is a hexadecimal constant, the length of the constant, rather than the length of the parameter being modified, is used when SDS makes the modification.

If type, length or scale qualifiers are used, each modifier is converted according to the resulting type, length and scale attributes of the parameter being modified. (Since instructions define their own length, the length of type I parameters is ignored.)

Verification of the modification is given by printing both the old value and the new value of the location modified. Verification may be suppressed by entering terse mode (see the TERSE option in the SET command description).

Examples:

```
MODIFY BETA(3) '87 96 84 2'
```

The first four locations starting with BETA(3) are modified to the constants 87, 96, 84, and 2, respectively.

```
MODIFY DELTA@T=X '0000003E'
```

DELTA is modified in hexadecimal format to the constant 0000003E.

```
MODIFY GR1 GR3
```

The contents of general register 3 are copied to general register 1. GR3 retains its original contents.

MTS

DEBUG COMMAND DESCRIPTION

Prototype: MTS [optional-command]

Action: Control returns to MTS command mode. The MTS command SDS returns control to debug command mode, from which the user can then resume debugging his program. An optional MTS command may be specified, which is executed in MTS command mode before control is given to the user.

Example: In the following example, the SDS user displays the general registers, returns to MTS, dumps his storage into a scratch file, and returns to SDS, which solicits for its next command. The status of the program being checked out has not changed and is restarted with the CONTINUE command.

```
DISPLAY GRS
MTS
$DUMP ON -TEMP
$SDS
CONTINUE
```

April 1974

QUALIFY

DEBUG COMMAND DESCRIPTION

Prototype: QUALIFY simple-parameter ...

Action: The type and scale attributes of a symbol are changed according to the type and scale modifiers appended to that symbol. The length attribute of a symbol may not be changed. The attributes of an instruction may not be changed.

Example:

```
QUALIFY ABC@T=X
```

The type attribute of ABC is changed to X (hexadecimal).

RESET

DEBUG COMMAND DESCRIPTION

Prototype: RESET

Action: All SDS operational parameters are reset to their default values. A list of these defaults, and the corresponding commands to change their values, is presented below.

Parameter	Default	Command
SDS command source	*MSOURCE*[1]	SET or INPUT
SDS output sink	*MSINK*[1]	SET or OUTPUT
Attention interrupt switch	ON	SET or ATTNTRP
Program interrupt switch	ON	SET or PGNTTRP
SDS prefix character	+	SET or PREFIX
Default type	X (hex)	SET or TYPE
Default length	4	SET or LENGTH
Default scale factor	0	SET or SCALE
Default relocation factor	0	SET or RF
Terse mode switch	OFF	SET or TERSE
At-mode prefix character	%	SET
Indirection character	$	SET
Modifier character	@	SET
Open-map character	*	SET
Blank-named common symbol	BLANK	SET
LINK/LOAD/XCTL switch	OFF	SET
Error-dump facility	OFF	SET

---

[1] Initially, the command source and output sink are *SOURCE* and *SINK*, respectively.

April 1974

RESTORE

DEBUG COMMAND DESCRIPTION

Prototype: RESTORE [simple-parameter ...]

Action: If no parameter is specified, the most recently entered global breakpoint or at-point set by the BREAK or AT command is removed from the user's program and the original instruction is restored. If one or more simple parameters are specified, the breakpoint or at-point at each location is removed and the original instruction is restored.

Example:

```
RESTORE LOC2
```

The breakpoint or at-point at location LOC2 is removed from the program.

RUN

DEBUG COMMAND DESCRIPTION

Prototype: RUN [simple-parameter ...]

Action: Control is transferred to the entry point of the program and execution is started. General registers 1, 13, 14, and 15 are set to the following values:

GR1 points to the system parameter list.
GR13 points to a system save area.
GR14 contains the return address.
GR15 contains the entry point address.

The other registers and the floating-point registers are set to zero. The entry point and the system parameter list may be changed by the ENTRY and PAR options of the SET command.

If a simple-parameter is specified, a local breakpoint is set at the location specified. This breakpoint code is an X'0004' which replaces the instruction. When the program encounters a local breakpoint, control returns to debug command mode. Local breakpoints are in effect only for the duration of the command, and are automatically erased before the user enters his next command.

For programs that are serially-reusable, that is, they are capable of being rerun several times without being reloaded, the RUN command may be used to restart the program at its entry point. Programs that are serially-reusable are either re-entrant or they do not modify their constant areas.

Example:

```
RUN LOC2
```

A local breakpoint is set at location LOC2 and execution of the user's program is initiated.

April 1974

SCAN

DEBUG COMMAND DESCRIPTION

Prototype: SCAN [{section|address1...address2|*} 'value']

Action: If section is specified, SDS searches through the named section in an attempt to find the value specified. The type and length attributes (and/or modifiers) of the section parameter determine how the value is converted for the search. The value given must be enclosed in primes.

If address1...address2 is specified, then address1 and address2 are the lower and upper bounds of the area to be searched. value is converted according to the type and length attributes of address1.

If * is specified, SDS searches through all loaded sections and defined dsects. value is converted according to the type and length attributes of *.

If the type specified is hexadecimal, character, or packed or zoned decimal, the length of the constant specified is taken as the length of the value to scan for; otherwise, the default type and length attributes are used for the scan if no TYPE and LENGTH modifiers are specified on the first parameter.

If no parameter is specified, the search resumes starting at the first location beyond the previous match from the previous SCAN command.

The search for the specified value is performed with respect to the appropriate boundary alignment of the value specified, that is, instructions are scanned for on halfword boundaries, character constants on byte boundaries, and so on.

Examples:

```
SCAN MAIN@T=E@L=8 '3.33762'
```

The section MAIN is scanned for the double-precision floating-point constant 3.33762.

```
SCAN 516100...5162DB@T=C 'OUTPUT'
SCAN
```

The region 516100...5162DB is scanned for the character constant OUTPUT. The remainder of the same region is scanned for a second occurrence of OUTPUT.

SDS

DEBUG COMMAND DESCRIPTION

Prototype: SDS

Action: If an at-point command list is being processed, the command list processing is terminated and control is returned to debug command mode. The SDS command should be given as the command preceding the END command in the at-point command list.

If the user is in debug command mode, no action is taken.

This command is not the same as the MTS $SDS command which is used to enter debug command mode from MTS command mode.

Example:

```
AT LOC5
DISPLAY A(1)...(9)
DISPLAY FRS
SDS
END
```

After the command list for the at-point at location LOC5 is processed, control is returned to debug command mode.

SET

DEBUG COMMAND DESCRIPTION

Prototype: SET keyword-parameter ...

Action: The SET command is used to alter the status of a number of SDS options, default attributes, or default characters. The valid keyword parameters are as follows:

ATPREFIX=char — The at-insertion mode prefix character becomes the character specified by char. The default is the "%" character.

ATTN={ON|OFF} — If the option is OFF, user attention interrupt exit routines set up by calls to the subroutine ATTNTRP are disabled. Consequently, SDS processes all attention interrupts. The option may be set to OFF before or after the user's program has called the subroutine ATTNTRP. The option may be set to ON to restore normal interrupt processing. The default is ON.

BLANK=chars — The blank-named common symbol used to refer to blank-named common sections becomes the character string specified by chars. The symbol may not begin with the indirection character ($), and may not contain any of the following characters: ()+-,.=' or the modifier character (@). The default value is BLANK.

ENTRY=loc — The entry point to the user's program is set to the location specified by loc. loc may be a symbolic, relative, or virtual address.

ERRORDUMP={ON|OFF} — If the option is ON and the user is running in batch mode, a symbolic dump is automatically given for abnormal program termination. The default is OFF.

INDCH=char — The indirection character becomes the character specified by char. The default is "$".

INPUT=FDname — SDS reads subsequent commands from the file or device specified by FDname. If an end-of-file is detected from the new

command stream, or if an attention interrupt, a program interrupt, or a breakpoint (not an at-point) is encountered, SDS returns to *MSOURCE* for its commands. The default is *SOURCE*.

LEN=i — The default length attribute is set to i, where i is an unsigned decimal integer between 1 and 256. Initially, the default length attribute is 4.

Logical I/O Units — Logical I/O unit assignments for the user's program may be given for both input and output units. This may be used to reposition files but does not affect the status of other types of pseudo-devices (such as *SOURCE* and the position of tapes).

MODCH=char — The modifier character becomes the character specified by char. The default is the "@" character.

OMAPCH=char — The "open-map" character becomes the character specified by char. The default is the "*" character.

OUTPUT=FDname — SDS writes subsequent output lines to the file or device specified by FDname. If an attention interrupt, a program interrupt, or a breakpoint (not an at-point) is encountered, SDS switches its output to *MSINK*. The default is *SINK*.

PAR=text — The system parameter list becomes the character string specified by text. Since text includes the remainder of the input line, the PAR option must be the last option specified on the SET command. A blank character is always added at the end of text.

PGNT={ON|OFF} — If the option is OFF, user program interrupt exit routines set up by calls to the subroutine PGNTTRP are disabled, SDS processes all program interrupts. The option may be set to OFF before or after the user's program has called the subroutine PGNTTRP. The option may be set to ON to restore normal interrupt processing. The default for this option is ON.

PREFIX=char — The SDS prefix character becomes the character specified by char. The default is the "+" character.

RF=constant — The unsigned hexadecimal constant defines the default relocation factor. This constant is added to every relative address which is not qualified by the @C= modifier. The default value for the relocation factor is 0.

SCALE=i — The default binary scale factor is set to i, where i is a signed or an unsigned decimal integer. Initially, the default scale factor is 0.

TERSE={ON|OFF} — If the option is ON, SDS enters terse mode and eliminates many confirmation and diagnostic messages. If the option is OFF, normal message processing occurs. The default is OFF.

TYPE=code — The default type attribute is set to code, where code is any of the single character SDS type codes described with the description of the TYPE keyword modifier. Initially, the default type attribute is X (hexadecimal).

XFR={ON|OFF} — If the option is ON, SDS intercepts all calls to the subroutines LINK, LOAD, and XCTL and returns to debug command mode. The modules specified in the subroutine call are loaded and the registers are set up for the execution of the loaded modules (in the case of LINK and XCTL). For XCTL, the calling program is unloaded and its symbols are purged from the SDS map. If the option is OFF, SDS does not intercept the subroutine calls. The default is OFF.

Examples:

```
SET LEN=8 TYPE=C TERSE=ON
```

This command sets the default length attribute to 8 bytes, the default type attribute to C (character), and sets terse mode ON.

```
SET SCARDS=INPUT SPRINT=OUTPUT PAR=EXEC
```

This command sets SCARDS and SPRINT to the files INPUT and OUTPUT, respectively, and sets the system parameter list to the character string EXEC.

STEP

DEBUG COMMAND DESCRIPTION

Prototype: STEP [i]

Action: The next i machine language instructions in the user's program are simulated before control returns to SDS. If i is not specified, only the next instruction is simulated.

Comments: If the user attempts to STEP past a branch instruction, the branch is taken as usual unless the program is transferring to a legal low-core symbol such as SCARDS or SPRINT. In this case, the routine is executed, not simulated, and stepping resumes at the return address. The instructions executed in the routine are not counted in the stepping count.

If the branch address is less than 300000, and does not correspond to the entry point of a legal low-core symbol, simulation is terminated and a warning message is printed. The user must restart his program with either a CONTINUE or GOTO command.

If STEP is used instead of RUN to initiate program execution, registers 1, 13, 14 and 15 are loaded with the appropriate values. (See the RUN command description.)

FORTRAN users should note that the STEP command specifies machine language instructions in the count. To step a specified number of FORTRAN instructions, the CONTINUE command should be used specifying temporary breakpoints.

Example:

```
STEP 10
```

The next 10 machine instructions in the program are simulated.

STOP

DEBUG COMMAND DESCRIPTION

Prototype: STOP

Action: SDS processing is terminated and control is returned to MTS command mode.

All loaded sections are unloaded; all loaded program symbol table information and the basic SDS work storage is released.

SYMBOL

DEBUG COMMAND DESCRIPTION

Prototype: SYMBOL simple-parameter ...

Action: The location specified by each simple parameter is printed in symbolic format. If no symbol table is present, the relative address and section name are printed. If this cannot be done, the corresponding virtual address is printed.

Example:

```
SYMBOL ALPHA ALPHA+4 $GR14 516020
```

This example displays the above locations in the following form:

```
ALPHA = ALPHA IN SECTION PROG
ALPHA+4 = VSYS IN SECTION PROG
$GR14 = LINK(8)+2 IN SECTION MAIN2
516020 = BETA IN SECTION SUBR
```

April 1974

USING

DEBUG COMMAND DESCRIPTION

Prototype: USING dsect-name address

Action: The dsect named by dsect-name may be assigned an address in two ways:

(1) address is a simple parameter which is used as a static base address for the dsect.
(2) address is a general register, the contents of which are used as a dynamic base address for the dsect (that is, the base address varies with the contents of the register).

A dsect may be redefined by subsequent USING commands.

If the dsect name occurs in more than one assembly, the @C modifier may be used to specify the desired dsect.

The DROP command may be used to undefine a dsect.

Examples:

```
USING DSECT1 GR1
```

The dsect DSECT1 becomes dynamically addressable by the contents of general register 1.

```
USING DSECT2 516200
```

The dsect DSECT2 becomes statically addressable by the base address 516200.

```
USING WORKAREA@C=SUBA $WADDR
```

The dsect WORKAREA from the assembly which contains the section SUBA becomes statically addressable by the current contents of the location WADDR.

April 1974

ABNORMAL CONDITIONS

The following paragraphs describe various abnormal conditions that may occur during the execution of a program, how the resulting interrupts are usually handled, and how the user may prepare to intercept them if they should occur.

There are several categories of interrupts, depending on what the program is doing. In this section, four categories are discussed.

PROGRAM INTERRUPTS

Fifteen different conditions cause a program interrupt (exception). These are the following:

Program Interrupt Cause	Interruption Code (hex)
Operation	1
Privileged operation	2
Execute	3
Protection	4
Addressing	5
Specification	6
Data	7
Fixed-point overflow	8
Fixed-point divide	9
Decimal overflow	A
Decimal divide	B
Exponent overflow	C
Exponent underflow	D
Significance	E
Floating-point divide	F

For a complete description of the meaning of these program interrupt conditions, see the IBM System/360 Principles of Operation, IBM form A22-6821. Normally, when one of these conditions occurs, the message

USER PROGRAM INTERRUPT. PSW=xxxxyyyy cxzzzzzz

is printed. The digits "yyyy" give the interruption code, and the digits "zzzzzz" give the location causing the interrupt. The first two bits of "c" give the instruction length code and the last two bits give the condition code. Program interrupt conditions 4 and 5 (protection and addressing exceptions) are caused by the user's program specifying an invalid address. An addressing exception is caused by an address specified outside the range

of virtual memory. A protection exception is caused by specifying an address in virtual memory which is not legal for the user's program to specify (usually an address reserved for the system supervisor). A protection exception may be an "imprecise" interrupt. (This occurs because the interrupt condition is discovered and flagged by the storage unit, not by the CPU, and hence the position of the CPU's location counter is imprecise.) An "imprecise" protection exception is indicated by the instruction length code (ILC) in the PSW being set to zero. After a program interrupt has occurred, the following happens:

(1) The general and floating-point registers are saved.
(2) A storage dump is given only in batch mode, if previously requested by one of the following MTS commands:

```
$ERRORDUMP
$SET ERRORDUMP=ON
$SET ERRORDUMP=FULL
```

(3) Control is returned to MTS command mode.

When a return is made to MTS command mode, the user can give any legitimate MTS command, including $RESTART, which causes the program to continue execution at the instruction following the one causing the program interrupt (assuming that the interrupt was not "imprecise").

A programmer can alter the normal processing of a program interrupt in several ways.

The BPI (branch on program interrupt) macro may be used in 360-assembler programs to specify a branch address to be taken when a specified type or class of interrupt type occurs. See the BPI macro description in Volume 3 for the complete description, calling sequences, and examples of usage.

The subroutine PGNTTRP (program interrupt trap) allows the user to specify his own exit routine which is transferred to upon the occurrence of a program interrupt. When the interrupt occurs and the exit is taken, the intercept is cleared so that another call to PGNTTRP is necessary to intercept the next program interrupt. PGNTTRP can be called directly from a 360-assembler program and indirectly through the subroutine RCALL in FORTRAN. See the PGNTTRP subroutine description in Volume 3 for the complete description, calling sequences, and examples of usage.

The subroutine SPIE (specify program interrupt exit), which is callable from a 360-assembler program, not only allows the user to specify an exit routine to be taken, but also provides the facility to specify the class of program interrupts for which this exit should be used. For any of the fifteen program interrupts not specified, the normal program interrupt procedure is followed. The specifications set up by a call to SPIE remain in effect until a subsequent call overrides them. This subroutine is normally called by using the SPIE macro. See the SPIE subroutine and macro descriptions in Volume 3 for the complete descriptions, calling sequences, and examples of usage.

When a program interrupt occurs, a check is first made to determine if any of the immediately following instructions are BPI macro instructions. If there is one, the type of program interrupt that occurred is compared with the type specified by the BPI macro. If there is a match, the condition code is set to reflect the interrupt that occurred and the branch is taken. If there is no BPI transfer made (either because there was no BPI instruction or because the program interrupt type did not match the BPI type), then a check is made to determine if a PGNTTRP (or SPIE) exit is active. If there is one, the exit is taken; otherwise, the program interrupt message is printed and a return is made to MTS command mode.

In PL/I, many asynchronous conditions are predefined as ON-conditions. Therefore, the user can determine by means of the ON statement what processing should occur upon the occurrence of one of these conditions. The following program interrupts are among these predefined conditions:

Program Interrupt	ON-condition Code
Fixed-point overflow	FIXEDOVERFLOW
Fixed-point divide	ZERODIVIDE
Exponent overflow	OVERFLOW
Exponent underflow	UNDERFLOW
Floating-point divide	ZERODIVIDE

Those asynchronous conditions that are not predefined cannot be intercepted. Those that cannot be intercepted and those that the user chooses not to intercept are handled by the PL/I library. See the IBM System/360 PL/I Reference Manual, IBM form C28-8201.

ATTENTION INTERRUPTS

Attention interrupts usually occur when a user is executing from a terminal and presses the ATTN or BREAK key. The system responds with

```
ATTN!
```

or, if a program was in executicn

```
ATTENTION INTERRUPT AT xxxxxxxx
```

where "xxxxxxxx" is the hexadecimal address of the point of interruption. The system returns to MTS command mode. Once in command mode, the user may give any legal MTS command. In particular, if the interrupt occurred during execution of a program, the program can be restarted from the location at which it was interrupted by issuing a $RESTART command.

The subroutine ATTNTRP (attention interrupt trap) allows the user to specify his own exit routine which is transferred to upon the occurrence of an attention interrupt. When the interrupt occurs and the exit is taken, the intercept is cleared so that another call to ATTNTRP is necessary to

intercept the next attention interrupt. ATTNTRP can be called directly from a 360-assembler program and indirectly through the subroutine RCALL in FORTRAN. See the ATTNTRP subroutine description in Volume 3 for the complete description, calling sequences, and examples of usage.

TIMER INTERRUPTS

Three different types of timer interrupts may occur while a program is executing. They are:

(1) Global time limit exceeded, which can happen only in batch mode, and is determined by the TIME parameter in the $SIGNON command.
(2) Local time limit exceeded, which can happen either in batch or conversational mode, and is determined by the TIME parameter in the $DEBUG, $LOAD, $RESTART, $RUN, and $START commands.
(3) Program specified timer interrupts, which are established by calls on certain system subroutines.

The first two types result in the system message

```
***GLOBAL TIME LIMIT EXCEEDED AT xxxxxxxx
```

or

```
***LOCAL TIME LIMIT EXCEEDED AT xxxxxxxx
```

where "xxxxxxxx" is the hexadecimal address of the point of interruption. These conditions cause a dump in batch mode if the ERRORDUMP option is specified. Neither of these interrupts can be intercepted by a program, but it is possible to determine the times at which they occur (via the GUINFO subroutine) and set up a program specified timer interrupt to occur before the global or local limit occurs. Thus, a program can arrange to print more useful diagnostic information in cases of infinite loops, and so on.

Program specified timer interrupts are set up by calls to the subroutines TIMNTRP, SETIME, GETIME, RSTIME, TWAIT, and TICALL which are described in Volume 3. These subroutines provide for the enabling and disabling of timer interrupts, and the specification of time intervals in several forms, including real time or task time, relative or absolute time, and in units of microseconds, timer units, or character string time of day.

INPUT AND OUTPUT ERRORS

The input and output of data by programs is handled by I/O subroutines called from the program. The I/O subroutines always provide a return code, the exact meaning depending on the file or device used in the operation. A description of the return codes that may occur with a particular file or

device is given in the appendix "I/O Routines' Return Codes" in the section "Files and Devices" in this volume.

In general, a return code of zero means successful completion of the input or output operation, and a return code of 4 means end-of-file-or-device for an input operation. Return codes greater than 4 normally signify an error condition and are not passed back to the caller, but instead cause an error comment to be printed and control to be returned to MTS command mode. The error comment usually describes the error condition and its location. When a return is made to command mode, any legal MTS command may be given.

There are two ways to suppress this error handling procedure and to make the calling program regain control. The subroutines SETIOERR and SIOERR allow the user to specify the location of a user supplied subroutine to be called upon the occurrence of an I/O error. Thus, when an I/O error occurs, the user supplied subroutine is called. This subroutine performs its tasks and returns to the I/O routine. The I/O routine then returns to the original calling program as if no error had occurred, except that the error code indicates which type of error happened. The error exit set up by SETIOERR or SIOERR remains in effect until overridden by a new call or until the program terminates normally. SETIOERR is for 360-assembler language users and SIOERR is for FORTRAN users. See the SETIOERR and SIOERR subroutine descriptions in Volume 3 for the complete description, calling sequences, and examples of usage.

The second way to suppress the error comment and return the error code to the calling routine is through the I/O modifier ERRRTN. If this modifier is specified in an I/O subroutine call and no SETIOERR-SIOERR interception has been established, then when an error occurs, the return code is passed back to the calling routine and no error comment is printed.

PL/I has predefined many of the I/O errors as ON-conditions. Thus, processing of these interrupts can be determined by ON statements.

April 1974

USING ERRORDUMPS AND LOAD MAPS

When running a program in batch mode, a hexadecimal dump and a load map may be useful tools in debugging a program. If the program ends abnormally and a dump has not been requested, only the program status word (PSW) and the general and floating-point registers may be obtained. This may not be enough information to find out where and why the program produced the error.

The user may use the MTS $ERRORDUMP command to obtain the contents of his virtual memory, as well as other information, if the program is terminated abnormally. If the program terminates normally, no dump is generated. Before running the program, the MTS command $ERRORDUMP or the command $SET ERRORDUMP=ON should be given. The MAP parameter to the $RUN command may be given to obtain a load map. The map is produced regardless of how the program is terminated.

In the example described below, the assembly language subroutine in the file TEST contains an intentional error for illustrative purposes. This subroutine is assembled into the file QUAD. The program in the file TESTCALL is a simple calling program to test the subroutine. This program is assembled into the file QUADCALL. The commands to assemble and run the sample program described below are:

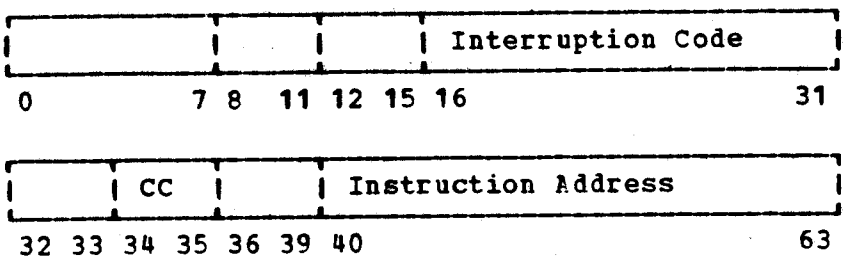
```
$RUN *ASMG SCARDS=TESTCALL SPUNCH=QUADCALL 0=*SYSMAC
$RUN *ASMG SCARDS=TEST SPUNCH=QUAD 0=*SYSMAC
$ERRORDUMP
$RUN QUADCALL+QUAD MAP
```

The map is printed immediately after the program has been loaded, and before execution of the program begins (see below). All the numbers in the map are hexadecimal. The "ENTRY=xxxxxx" is the address of the first instruction to be executed. "SIZE=xxxxxx" is the sum of the lengths of all the loaded control sections. The NAME is the name of the external symbol; its VALUE is the actual address assigned to the external symbol. If the value is six dashes, the external symbol is undefined. "T" gives a type indication for the symbol. This column may contain "*" for system symbols, "C" for program common sections, "D" for symbols defined with a DEF control record, "P" for pseudo register storage, or blank, for all others. Every symbol which is the name of a control section also has a relocation factor printed in its entry. This is the number which must be added to the given address in the assembly listing to get the virtual memory address.

The first line of the dump contains the date and time of the dump initiation. The second line is the program status word (PSW). This contains information about the program; a full description may be found in the section titled "Status Switching" in IBM System/360 Principles of Operation. A summary is given in the IBM System/360 Reference Data card (usually referred to as "the green card"). Only those parts of the PSW most

commonly used for debugging are explained here. The last three bytes (last six hexadecimal digits) are the address in virtual memory of the instruction following the one being executed at the time of termination. In the case of a program interrupt, the second and third bytes (last four hexadecimal digits of the first word) contain the interruption code. (It is sufficient to look only at the third byte.) The interruption code explains the reason why the program interrupt occurred. It may be that an instruction referenced an address that is outside the user's virtual memory space (interrupt code 5) or that an illegal operation code was encountered (interrupt code 1). The interrupt codes are summarized in the section "Abnormal Conditions" in this volume, and the causes of various types of interrupts are listed in the section titled "Interruptions" in IBM System/360 Principles of Operation. Another item which is occasionally useful is the condition code. This is set by comparisons and a number of other instructions. It occupies bits 2 and 3 of the fourth byte[1] of the PSW (last two bits of the first hexadecimal digit in the second word). Its values are 0, 1, 2, and 3 and these correspond to values of 8, 4, 2, and 1 respectively in the mask of the branch-on-condition instruction.

Program Status Word



See the Principles of Operation for the complete description of the Program Status Word.

Following the PSW is a one line heading and two lines displaying the contents of the general registers. The first line shows general registers 0 to 7, and the second line shows general registers 8 to 15. Information which may be verified from the contents of the general registers includes the correct value of the base register, the last location from which a subroutine call was made (usually in general register 14), the location of the current save area (general register 13), the return code from the last called subroutine (usually in general register 15), and the values of computations involving general registers.

After the general registers, the four floating-point registers (0, 2, 4, and 6) are displayed. These may be used to check on values produced using the floating-point instructions.

---

[1] Bytes of the PSW are numbered 0-7, and bits within a byte are numbered 0-7 (from the left).

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT      LINE #   SOURCE STATEMENT

                                      1       1.000          PRINT NOGEN
000000                                2       2.000          START
                                      3       3.000          ENTER 12,SA=SAVE
000014 5890 C0A8            000A8    11       4.000          L     9,=F'-1'
000018 5810 C0AC            000AC    12       5.000          L     1,=A(PARLIST)
00001C 58F0 C0B0            000B0    13       6.000          L     15,=V(QUAD)
000020 05EF                          14       7.000          BALR  14,15
                                     15       8.000          EXIT  0
000032 0000
000034 00000048                      21       9.000 PARLIST  DC    A(A)
000038 0000004C                      22      10.000          DC    A(B)
00003C 00000050                      23      11.000          DC    A(C)
000040 00000054                      24      12.000          DC    A(ANS1)
000044 00000058                      25      13.000          DC    A(ANS2)
000048 41200000                      26      14.000 A        DC    E'2.0'
00004C 41500000                      27      15.000 B        DC    E'5.0'
000050 C1C00000                      28      16.000 C        DC    E'-12.0'
000054                               29      17.000 ANS1     DS    E
000058                               30      18.000 ANS2     DS    E
00005C                               31      19.000 SAVE     DS    18F
                                     32      20.000          END
0000A8 FFFFFFFF                      33                      =F'-1'
0000AC 00000034                      34                      =A(PARLIST)
0000B0 00000000                      35                      =V(QUAD)
```

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT      LINE #   SOURCE STATEMENT

                                      1       1.000          PRINT NOGEN
000000                                2       2.000 QUAD     START
                                      3       3.000          ENTER 12,SA=SAVE
000014 9826 1000            00000    11       4.000          LM    2,6,0(1)      ADDRESSES OF ARGUMENTS
000018 7800 3000            00000    12       5.000          LE    0,0(,3)       B
00001C 3C00                          13       6.000 Q1       MER   0,0           B*B
00001E 7820 C110            00110    14       7.000          LE    2,=E'4'
000022 7C20 2000            00000    15       8.000          ME    2,0(,2)       4*A
000026 7C20 4000            00000    16       9.000          ME    2,0(,4)       4*A*C
00002A 3B02                          17      10.000 Q2       SER   0,2
00002C 4740 C078            00078    18      11.000          BM    COMPLEX
000030 7000 C0C4            000C4    19      12.000          STE   0,DISCR
                                     20      13.000          CALL  SQRT,DISCR    SQRT(B**2-4*A*C) IN PRO
00004A 7820 2000            00000    32      14.000 Q3       LE    2,0(,2)       A
00004E 3A22                          33      15.000          AER   2,2           A+A
000050 3340                          34      16.000          LCER  4,0           -SQRT(B**2-4*A*C)
000052 7B40 3000            00000    35      17.000          SE    4,0(,3)       -B-SQRT(B**2-4*A*C)
000056 3D42                          36      18.000 Q4       DER   4,2           (-B-SQRT(B**2-4*A*C))/(2*A)
000058 7040 9000            00000    37      19.000          STE   4,0(,9)       SECOND ROOT
00005C 7B00 3000            00000    38      20.000          SE    0,0(,3)       -B+SQRT(B**2-4* A*C)
000060 3D02                          39      21.000          DER   0,2           (-B+SQRT(B**2-4*A*C))/(2*A)
000062 7000 5000            00000    40      22.000 Q5       STE   0,0(,5)       FIRST ROOT
                                     41      23.000          EXIT  0
                                     47      24.000 COMPLEX  SERCOM  ' *** EQUATION HAS COMPLEX ROOTS'
                                     57      25.000          EXIT  4
0000C4                               63      26.000 DISCR    DS    E
0000C8                               64      27.000 SAVE     DS    18A
                                     65      28.000          END
000110 41400000                      66                      =E'4'
000114 00000000                      67                      =V(SERCOM)
```

Figure 1: Sample Assembler Program

```
$ERRORDUMP
$RUN QUADCALL+QUAD MAP

... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ..

ENTRY = 5001D8  SIZE = 0001CC

NAME      VALUE  T RF     NAME      VALUE  T RF     NAME      VALUE  T RF

SERCOM    221B3C *         LCSYMBOL 222B10 *     ' <EFL>     275000 *
SQRT      278E18 *                  5001D8  5001D8 QUAD      500290  500290
<SYNTAB>  501000 *501000

... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ..


 DATE  01-26-73       TIME  18:26.35

 PSW    =  07150006 905002EC

         GENERAL REGISTERS 0...15
                  F72A8D61   A05002D0   00500220   00500224   00500228   0050022C   00500230   00000000
                  00000000   FFFFFFFF   00000000   00000000   00500290   00500358   605002DA   00000000

         FLOATING POINT REGISTERS
                  41B00000 00000000     41A00000 00000000     C1A00000 00000000     00000000 00000000


                  STORAGE INDEX: 0000
   LA       RA
500000             00000000  00000000  81818181  81818181    81818181  81818181  81818181  81818181  *........AAAAAAAAAAAAAAAAAAAAAAAA*
...... ......     ........
500100 000100    80500000  00211654  005001D8  81818181    00000000  00500234  00211654  005001D8  *.&........&.QAAAA.....&........&.Q*
500120 000120    00000000  00500100  00000000  00000000    00000000  00000000  00000000  00000000  *.....&.............................*
500140 000140    00000000  00000000  00000000  00000000    00000000  81818161  1C818181  81818181  *.....................AAAA.AAAAAAAA*


CSECT:               ADDR: 5001D8  RELOC: 5001D8  LENGTH: 0000B8  STORAGE INDEX: 0080
  LA     RA
5001C0                                                                 90ECD00C  18CF41F0                  *.........0*
5001E0 000008    C05C50F0  D00850D0  F00418DF  5890C0A8    5810C0AC  58F0C0E0  05EF58D0  D00498EC  *.*&0..&.0......Y.....0.........Q.*
500200 000028    D00C92FF  D00C1BFF  07FE0000  00500220    00500224  00500228  0050022C  00500230  *..K..........&....&...&...&...&..*
500220 000048    41200000  41500000  C1C00000  40000000    00000000  000065EF  0050010C  00500358  *.....&..A...  .............&...&..*
500240 000068    405001FA  00500290  00000000  0050020C    00000000  00000000  00000000  00000000  * &...&........&...................*
500260 000088    00000000  00000000  00000000  FFFFFFFF    00000000  00000000  005001D8  40404040  *...............................&.Q    *
500280 0000A8    FFFFFFFF  0050020C  00500290  E2C1E2E5                                            *.....&...&..SASV*


CSECT:  QUAD         ADDR: 500290  RELOC: 500290  LENGTH: 000118  STORAGE INDEX: 0080
  LA     RA
500280                                                                 90ECD00C  18CF41F0  C0C850F0  D00850D0  *.........0.H&0..&.*
5002A0 000010    F00418DF  98261000  78003000  3C007820    C1107C20  20007C20  40003B02  4740C078  *0...Q.............&...B. .... ..*
5002C0 000030    7000C0C4  47F0C03C  00278E18  4510C044    00500354  58F0C038  05EF7820  20003A22  *...D.0.............&...0..........*
5002E0 000050    33407B40  30003D42  70409000  78003000    3D027000  500058D0  D00498EC  D00C92FF  *. @ .....  ..@.......&.....Q...K.*
500300 000070    D00C1BFF  07FE0700  58F0C114  4510C0AE    00500318  0050031C  00000000  001F405C  *..........0A......&...&........ **
500320 000090    5C5C40C5  D8E4C1E3  C9D6D540  C8C1E240    C3D6D4D7  D3C5E740  D9D6D6E3  E20005EF  *** EQUATION HAS COMPLEX ROOTS...*
500340 0000B0    58D0D004  98ECD00C  92FFD00C  41F00004    07FEC1E2  42790000  C4C1E340  00500234  *....Q...K....0....AS....DAT .&..*
500360 0000D0    40D4E3E2  FF5002DA  00278E18  00000000    A05002D0  00500220  00000000  58000000  * MTS.&.........&...&..........*
500380 0000F0    00000000  000067DD  00000000  00000000    00000000  00000000  41000001  00000000  *.....................&.........*
5003A0 000110    41400000  00221B3C                                                                *. ......*


CSECT:  <SYNTAB>   ADDR: 501000  RELOC: 501000  LENGTH: 002000  STORAGE INDEX: 0000
  LA     RA
501000             00002000  81818181  00501050  A0211462    00001000  00000000  00500170  A02115EE  *....AAAA.&.&...B.........&......*
501020 000020    50210EF8  00400000  00500160  00000000    00220096  00211BA0  005001D8  0021B5CC  *&..&. ...&.-.............&.Q....*
501040 000040    00214954  00213954  40000007  81818181    81818181  00501000  81818181  40002A38  *.........  ...AAAAAAAA.&..AAAA ...*
501060 000060    00000000  00000000  00501DD4  B0001E14    00501458  00000003  00501420  00501418  *.........&.H.....&.......&....&..*
501080 000080    00501000  00501000  00000000  00501538    00003000  8000155A  60211ADE  000033A4  *.&...&.......&..........-.....0*
5010A0 0000A0    00002000  00403D3C  00403CD8  40211AC6    00400000  00000000  00403C90  00401800  *..... ... .Q ..D. ..... ... ...*
5010C0 0000C0    00402000  00400000  0021C10C  00214954    00213954  00501408  0050123E  00002B88  *. ... ....A..........&...&.....H*
5010E0 0000E0    00501308  0050123C  005010F0  00501234    00004015  005011FC  005011E4  00501320  *.&...&...&.0.&.... ..&...&.U.&..*
501100 000100    00003030  00002000  00001000  00000000    00501458  000061D0  005001D8  00500290  *.....................&.......&.Q.&..*
501120 000120    005014B8  00501234  00501318  00000010    00000001  00000020  00003000  90001836  *.&...&...&.......................*
501140 000140    00000000  0020F5EE  00211BA0  81818181    81818181  0000003E  00275124  005014D8  *.....5.....AAAAAAAA.........&.Q*
501160 000160    005014B8  0020F5EE  81818181  81818181    81818181  81818181  00000000  B000167A  *.&.Q..5.AAAAAAAAAAAAAAAA........*
501180 000180    00000001  00001970  00000000  00501538    C8000408  0020F296  0020F5EE  00211BA0  *.............&..H.....2O..5.....*
5011A0 0001A0    00000080  0021B5CC  00500290  00000000    00000118  81818181  81818181  005028B4  *.........&..........AAAAAAAA.&..*
5011C0 0001C0    00503000  00500290  005001D8  00000000    0021F71E  00221B6C  0020F296  0020F5EE  *.&...&...&.Q......7....%..2O..5.*
5011E0 0001E0    00222096  00000000  00503000  00270749    00503B6F  00000200  00002AD4  40000007  *.........&........&.?.......H ...*
501200 000200    07000000  00000007  00000000  00000000    00000000  000001CC  00000118  00000118  *.....................&...........*
501220 000220    00000000  0000197C  00000080  000000A0    81818181  00001B58  00010000  001A0047  *.......9.........AAAA.....&.....*
501240 000240    8181D8E4  C1C44040  4040D8E4  C1C44040    40400004  00000000  00000000  00000000  *AAQUAD     QUAD     ..............*
501260 000260    00000000  00000007  00050000  00000000    00000000  00000000  00000000  00000006  *.....................&...........*
501280 000280    00000000  00030000  00000000  00000000    00000000  00000000  00000000  00010000  *.....................&...........*
5012A0 0002A0    00000000  00000000  00000000  00000000    00000000  00000000  00000000  00020000  *.....................&...........*
5012C0 0002C0    00000000  00000000  00000000  E2D8D9E3    40404040  00278218  00000000  00008800  *............SQRT  ..........H.*
5012E0 0002E0    00000000  00000000  00000000  005003A4    00501478  00000000  00501000  F0F0F5F0  *.....................&.U.&.....0050*
501300 000300    F1F0F0F0  0F818181  40D3C4E3    40404040    40404040  40404040  40404040  40404040  *1000.AAA LDT                    *
501320 000320    F1F87AF5  F24BF1F9  40404040    404040C1    E2D4C740  F7F2D1E4  D3F94040  40F1F87A  *#18:52.19       ASHG 72JUL9    18:*
501340 000340    F2F54BF3  F94040F1  40C6C5C2    40F7F340    40404040  F0F0F1F1  81818181  81818181  *25.39  1 FEB 73     0011AAAAAAAA*
...... ......     ........
501400 000400    81818181  81818181  40404040    40404040    40404040  40404040  40404040  40404040  *AAAAAAAA                        *
...... ......     ........
501440 000440    40404040  40404040  40404040    40404040    40404040  40404040  4CE2E8D4  E3C1C26E  *                          <SYNTAB>*
501460 000460    00501000  00501000  0400A108    00000000    80000000  00000000  D3C3E2E8  D4C2D6D3  *.&...&.....................LCSYMBOL*
501480 000480    00222B10  00000000  00009900    00800000    00000000  00000000  40404040  40404040  *.........R.............          *
5014A0 0004A0    005001D8  005001D8  0017B148    00800000    00000000  00000000  D8E4C1C4  40404040  *.&.Q.&.Q.....................QUAD    *
5014C0 0004C0    00500290  00500290  0023B108    00800000    00000002  00000000  E2D8D9E3  40404040  *.&...&.....................SQRT    *
5014E0 0004E0    00278E18  00000000  00009900    00800000    00000002  00000000  E2C5D9C3  D6D44040  *.............R.............SERCOM  *
501500 000500    00221B3C  00000000  00009900    00800000    00000003  00000000  4CC5C6D3  6E404040  *.............R.............<EFL>   *
501520 000520    00275000  00000000  00009900    00800000    00000000  00000000  81818181  81818181  *..&........R.....................AAAAAAAA*
...... ......     ........
502E80 001E80    81818181  81818181  81818181  81818181    81818181  81818181  81818181  005003A4  *AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.&.U*
502EA0 001EA0    005014F8  0000001C  005003A4  005014F8    0000001C  005802C8  005014D8  0000001C  *.&.8.....&.U.&.8.....&.H.&.Q....*
502EC0 001EC0    81818181  81818181  81818181  81818181    81818181  81818181  81818181  81818181  *AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA*
...... ......     ........
502FE0 001FE0    81818181  81818181  81818181  81818181    81818181  81818181  81810006  00050004  *AAAAAAAAAAAAAAAAAAAAAAAAAA......*
```

Figure 2: Sample Program Run, Map, and Dump

Finally, those parts of virtual storage being used by the program are displayed. (This does not include the parts of MTS being used by the program.) The number at the left hand side of each line is the address of the first byte displayed on the line. (At the beginning of a block of storage, the first bytes of the line may not be printed.) Each line is divided into eight four-byte (that is, eight hexadecimal digits) words.

Included here is an example program and the map and dump which it produced. Figure 1 shows the assembler output; and figure 2 shows the $RUN command, map, interrupt message, and dump. The program is a subroutine named QUAD which takes five arguments. The first three arguments are coefficients A, B, and C of the quadratic equation

```
A*X**2+B*X+C=0
```

The fourth and fifth arguments contain the real roots of the quadratic equation when the subroutine returns. Complex roots are not permitted. The object deck for QUAD is in the file QUAD, and there is an object deck for a main program (with a blank name) to test QUAD, in the file QUADCALL. Since there was a program interrupt, there is obviously an error. The interrupt code in the PSW is 6, a specification error. This type of interrupt may be caused by an alignment error, an odd register number when an instruction requires an even register number, etc. The Principles of Operation summarizes the reasons why each of the interrupt codes might occur. The interrupt address in the last three bytes of the PSW is 5002EC. Since 5002EC lies between the values 500290 and 501000, the interrupt occurred in QUAD. Subtracting the relocation factor for QUAD (which in this case happens to be the same as the value) from 5002EC gives 5C. In the assembler listing, 5C is the address of the SE instruction in statement 38. Since this is the instruction after the one being executed at the time of the interrupt, the instruction which caused the trouble is in statement 37 (unless some other instruction branches to statement 38).

All reasons for a specification interrupt except alignment can be quickly eliminated. To check on alignment, it is necessary to compute the address from the base, displacement, and index. There is no index and the displacement is zero; so the entire address is the contents of general register 9. The general register display in the dump shows that register 9 contains FFFFFFFF. This is certainly not divisible by 4; and therefore, it does not produce the proper alignment for a floating-point short instruction. Either the contents of the register are wrong or the base specification is wrong. The program is supposed to be storing the value of the second root. Looking at statement 11 in the assembler listing, it is seen that the addresses of the five arguments are in general registers 2 through 6. Therefore, the address of the fifth argument, which is the storage location of the second root, is in register 6, not register 9. Changing the instruction in line 37 to

```
STE   4,0(,6)
```

should correct the problem.

Before rerunning the program, the user should check that everything is correct up to that point, to avoid the "toss it in, maybe it'll work" syndrome. Looking first at register 14, it is seen from the map that 5002DA is in QUAD. Subtracting the relocation factor gives 4A. The address 4A in the assembler listing is that of the instruction after the call on SQRT. Consequently, SQRT was the last subroutine called. SQRT does not produce a return code; and therefore, register 15 still has the address of SQRT.

It can be verified that register 13 points to the save area in QUAD. The QUAD save area can be used to find the save area of the calling program. The convention for the save area is that the second word points to the previous save area, that is, the save area of the calling program. Looking at location 500358 (which is the address contained in register 13) in the dump, it can be seen that the previous save area is at location 500234. From the map, this is in the program with a blank name, or in other words, the test calling program.

Having found the previous save area, the user can now determine what the arguments were for QUAD. Registers 14 through 12 at the time of entry into QUAD are stored starting with word 4 of the save area. Therefore, register 1, which points to the list of argument addresses, is in word 7. This list address is 50020C. Looking at 50020C, it is seen that the five argument addresses are 500220, 500224, 500228, 50022C, and 500230. To put it another way, the value of A is at 500220, the value of B is at 500224, and the value of C is at 500228. Checking these locations in the dump shows that

A=2.0 B=5.0 C=-12.0

At the time of the interrupt, the value of the second root is in floating-point register 4. From the quadratic formula, the solutions for the above values of A, B, and C are 1.5 and -4.0. The contents of floating-point register 4 is -4.0; and therefore, the second root is correct. Likewise, the values of the discriminant and 2A can be verified from floating-point registers 0 and 2 respectively.

April 1974

PROJECTACCOUNT

INTRODUCTION

PROJECTACCOUNT allows project directors and instructors to distribute money, permanent disk space, and terminal and plotting time, as they wish, to various signon ID's belonging to their project or class. Also, the expiration time for individual signon ID's may be changed. These distributions are limited by maximums set for the entire project or class rather than having each signon ID with its own relatively fixed maximums. In addition, the amounts used and the maximums for any given signon ID may be displayed, as well as the amounts allocated compared with the maximums for the project and the totals for the project as a whole.

Before a project can use PROJECTACCOUNT, one of the signon ID's belonging to the project must be authorized to use the program. Only one signon ID is authorized per project. The authorization and setting of maximums for the project and this authorized signon ID can be accomplished by contacting the business office at the Computing Center.

In the examples which follow, it is assumed that the project number is G0335006, the signon ID authorized to use PROJECTACCOUNT is S001, and the other ID's belonging to the project are S002, S003, and S004. The maximum amount of money for the project is $550, and the maximum and used amounts of money for each signon ID are as follows:

ID	MAXIMUM $	USED $
S001	100	25
S002	100	10
S003	100	125
S004	100	0

The examples follow each other logically; that is, the result of each example is presumed to be the starting point for the next example. With the exception of the sections where the complete heading is given, the one line heading in each example is for convenience only and is not part of the actual output.

The discussion of the commands, signon ranges, and keywords assumes operation from a terminal. Batch operation involves only minor differences, and a batch job to produce the examples in this section appears at the end of this writeup.

To start the program after signing on to MTS using the authorized signon ID for the project, enter the MTS command

```
$RUN *PROJECTACCOUNT
```

after the "*" is printed. After execution begins, the program responds with

```
MTS ACCOUNTING MAXIMUM MAINTENANCE FOR PROJECT G0335006
```

and a question mark prefix is printed indicating that the program is ready for input.

COMMANDS

ADDing Money to a Signon ID

If $100 is to be added to the maximum charge for S003, enter

```
ADD S003,CHARGE=100
```

ADD is called the *command*. A *delimiter*, which is one or more blanks or commas, in any order, must follow the command. 100 is called the *value* of the keyword. A keyword and its value are always separated by an "=". The combination of the keyword "=" and the value is called a *keyword parameter*. Keyword parameters are separated from the signon range and from each other by delimiters.

After the $100 has been added to S003, the program responds with

```
  SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
        (DOLLARS)     (PAGES)   (HRS:MIN)  (HRS:MIN) DATE/TIME
 (MAXIMUM AMOUNT IS ABOVE USED OR CURRENT AMOUNT.  NC=NO CHANGE ON.)

  S003      200.00          10   IGNORED    IGNORED  12-31-70
            125.00           2      0:00       0:00     24:00
```

and another question mark prefix. The first line after the three line heading shows the new maximums for S003, and the second line shows the amount already used ($125). If this is all that is desired, *producing an end-of-file condition on input terminates execution of the program*.

SUBTRACTing Money from a Signon ID

While ADD causes the program to add the values of the keyword parameters to the current maximums, SUBTRACT causes the program to subtract the values of the keyword parameters from the appropriate current maximums. To reduce the maximum amount of money for S003 by $10, enter

```
SUBTRACT S003,CHARGE=10
```

The program responds with

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003      190.00          10   IGNORED    IGNORED  12-31-70
          125.00           2      0:00       0:00   24:00
```

and a question mark prefix. If the result of the subtraction had been negative, the maximum for the signon ID would have been set to zero.

MODIFYing Money for a Signon ID

MODIFY causes the program to replace the current values of the appropriate maximums with the values of the keyword parameters. To set the maximum amount of money for S002 to $15, enter

```
MODIFY S002,CHARGE=15
```

The program responds with

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S002      15.00           2   IGNORED    IGNORED  12-31-70
          10.00           0      2:00       0:00   24:00
```

EQUALIZing Money for a Signon ID

EQUALIZE causes the program to add the values of the keyword parameters to the appropriate current used amounts (as opposed to the current maximums for ADD) and replace the corresponding maximums with the result. Thus, to give S003 $25 more than has been used, enter

```
EQUALIZE S003,CHARGE=25
```

The program responds with

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003      150.00          10   IGNORED    IGNORED  12-31-70
          125.00           2      0:00       0:00   24:00
```

This command is useful for giving all the students in a class the same amount of money with which to do a new problem.

EXPIRing Money for a Signon ID

The command EXPIRE, without any keyword parameters, is the same as MODIFY with the keyword parameters CHARGE=0, DISK=0, TERMINAL=0, PLOT=0, EXPIRE=03-01-00, and NOCHANGE=ON. (See the "Keywords" section.) That is, all maximums are set to zero, the signon ID is expired, and "no change" is set on. Thus

```
EXPIRE S002
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S002      0.00             0       0:00       0:00 07-07-70
  NC     10.00             0       2:00       0:00   16:25
```

Any keyword parameters given with the command EXPIRE override the values of the corresponding implied keyword parameters. Therefore,

```
EXPIRE S004,CHARGE=95
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S004     95.00             0       0:00       0:00 07-07-70
  NC      0.00             0       0:00       0:00   16:25
```

Note that NOCHANGE=OFF does not override NOCHANGE=ON. (See the section "Changing NOCHANGE".)

Obtaining the STATUS of a Signon ID

The command STATUS enables one to print the maximum and used amounts for a signon ID.

```
STATUS S001
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S001    100.00             0   IGNORED    IGNORED 12-31-70
         25.00             0       0:00       0:00   24:00
```

Note that these are the amounts as of the last signoff of the signon ID. All keyword parameters except HEADING are ignored by STATUS. See also the section "PROJECT".

CONTINUing with a Different Signon ID

The command CONTINUE, without any keyword parameters, does the same thing as the previous command but uses the new signon range. It does not produce a new heading unless the keyword HEADING (see the section "Producing a HEADING") is explicitly used with the CONTINUE command. Thus if

```
ADD S002,CHARGE=5,NOCHANGE=OFF
```

which produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S002        5.00           0       0:00       0:00 07-07-70
           10.00           0       2:00       0:00   16:25
```

is followed by

```
CONTINUE S004
```

the program responds with

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S004      100.00           0       0:00       0:00 07-07-70
            0.00           0       0:00       0:00  16:25
```

Any keyword parameters given with CONTINUE override the corresponding keyword parameters used with the previous command. Thus, after the above sequence of ADD and CONTINUE

```
CONTINUE S002,CHARGE=10
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S002       15.00           0       0:00       0:00 07-07-70
           10.00           0        2:00       0:00   16:25
```

Note that NOCHANGE=OFF does not override the keyword parameter NOCHANGE=ON. (See the section "Changing NOCHANGE".)

Determining Lost PASSWORDs

The command PASSWORD enables project directors to display the current password for any signon ID belonging to the project. To display the password for S004, enter

```
PASSWORD S004
```

to produce

```
S004 PASSWORD IS X95MPS (E7F9F5D4D7E2)
```

The characters in parentheses are the hexadecimal representation of the password.

All keyword parameters are ignored by PASSWORD. Also, unlike other commands, PASSWORD must be specified for a single ID. Any attempt to use a block or ENTIRE as the signon range causes an error message to be printed.

OTHER SIGNON RANGES

Blocks

So far, the signon range has been a single signon ID. It may, however, refer to a group of signon ID's. For a specific, contiguous block of signon ID's belonging to the project, the signon range is the first signon ID followed by a delimiter, three dots (periods), an optional delimiter, and the last signon ID. For example,

```
STATUS S002,...,S004
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S002      15.00             0       0:00       0:00 07-07-70
          10.00             0       2:00       0:00   16:25

S003     150.00            10    IGNORED    IGNORED  12-31-70
         125.00             2       0:00       0:00   24:00

S004     100.00             0       0:00       0:00 07-07-70
           0.00             0       0:00       0:00   16:25
```

The incrementing of signon ID's within a block is defined as the collating sequence for the IBM 360 computer, that is, alphabetic letters come before the numbers 0 through 9. The first signon ID should be less than or equal to the last signon ID.

ENTIRE

If all of the signon ID's for the project are desired, the word ENTIRE may be used for the signon range. For example,

```
MODIFY S001,NOCHANGE=ON
SUBTRACT ENTIRE,CHARGE=10
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S002       5.00             0       0:00       0:00 07-07-70
          10.00             0       2:00       0:00   16:25

S003     140.00            10    IGNORED    IGNORED  12-31-70
         125.00             2       0:00       0:00   24:00

S004      90.00             0       0:00       0:00 07-07-70
           0.00             0       0:00       0:00   16:25
```

Note that S001 is not listed. The MODIFY command which sets NOCHANGE to ON is used to prevent money from being subtracted from the master signon. This is usually desirable as the master signon should only be used for running *PROJECTACCOUNT.

PROJECT

The above signon ranges can be used with any command. PROJECT can be used only with STATUS. Entering

```
STATUS PROJECT
```

produces

```
SIGNON  CHARGE  DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
PRJ       550.00          50   IGNORED    IGNORED  12-31-70
          340.00          10      2:00       0:00     24:00

REM       210.00          40   IGNORED    IGNORED
           52.50          10      0:00       0:00

TOT       335.00          10      0:00       0:00
          160.00           2      2:00       0:00
NOCHANGE ON FOR   1 AND OFF FOR   3 OF THE   4 PROJECT ID'S
```

This listing needs some explanation. The first line, labeled PRJ for "PROJECT", shows the maximums for the project. Note the $550 under charge. The second line shows the cumulative amounts for the project. Since MTS permits a user to finish, once he is signed on, a maximum may be less than the corresponding used or current amount. This same condition can also be produced by using PROJECTACCOUNT. Therefore, the cumulative amounts for the project are computed by summing the larger of the maximum and used or current amounts for each signon ID. (See also "Miscellaneous" regarding disk space.) Referring to the listings in the sections "Changing EXPIRation Time" and "ENTIRE", the cumulative amount of money is computed as follows: the larger of the maximum and used amounts for S001 is $100. The larger amount for S002 is $10, and the larger amounts for S003 and S004 are $140 and $90, respectively. Adding these amounts together produces the sum of $340.

The third line, labeled REM for "REMAINING", shows the results of subtracting the cumulative amounts (second line) from the maximums (first line) for the project. These differences are the amounts which can still be distributed to the signon ID's belonging to the project. In this case, $550-$340=$210 can still be added to individual signon ID's. The fourth line shows the amounts which should be added to each signon ID with "no change" currently off (see "Changing NOCHANGE") in order to evenly distribute the remaining amounts. That is, the values in the fourth line are the results of dividing the values in the third line by the number of ID's with "no change" off ($210/4=$52.50). Note that fractional or negative amounts cannot be used as keyword values.

The fifth line, labeled TOT for "TOTALS", shows the sums of the maximums for the signon ID's. Thus, the sum of the maximums of $100, $5, $140, and $90 for S001 through S004 is $335. The sixth line shows the sums of the used or current amounts for the signon ID's.

The seventh line states the number of signon ID's with "no change" on, the number of signon ID's with "no change" off, and the number of signon ID's belonging to the project (four in this case).

KEYWORDS

Changing Maximum CHARGE

The keyword CHARGE has been shown to specifiy changes in dollars of the maximum amount of money permitted to a signon ID. Other keywords are used to specify other maximums and control functions. All keyword parameters come after the signon range and may be listed in any order. They are separated from the signon range and from each other by delimiters. All numeric keyword values must be entered as unsigned integers without decimal points, that is, there may be no fractional or negative keyword values.

Changing Maximum DISK Space

The keyword DISK (or FILE) is used to specify changes in pages of the maximum disk space permitted to a signon ID. To add 5 pages of disk space to S003, entering

```
AID S003,DISK=5
```

produces

```
SIGNON  CHARGE   DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003      140.00         15   IGNORED    IGNORED  12-31-70
          125.00          2      0:00       0:00     24:00
```

The second line of the listing under DISK SPACE is the amount of disk space (2 pages) currently being used by S003.

Changing Maximum TERMINAL Time

Minutes

The keyword TERMINAL is used to specify changes in minutes of maximum terminal time permitted to a signon ID. To restrict S002 to two and a half hours of terminal time, entering

```
MODIFY S002,TERMINAL=150
```

produces

```
 SIGNON  CHARGE   DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
 S002       5.00           0       2:30       0:00 07-07-70
           10.00           0       2:00       0:00   16:25
```

If the project is permitted unlimited terminal time, subject only to available funds, signon ID's belonging to the project may also be permitted unlimited terminal time. Unlimited terminal time is indicated in the listing by the word IGNORED in place of the maximum. Unlimited terminal time may be set by using the word IGNORE as the value of the keyword TERMINAL. Thus to give S002, unlimited terminal time, entering

```
MODIFY S002,TERMINAL=IGNORE
```

produces

```
 SIGNON  CHARGE   DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
 S002       5.00           0   IGNORED       0:00 07-07-70
           10.00           0       2:00       0:00   16:25
```

When the value is IGNORE, the behavior is the same for each of the commands ADD, EQUALIZE, EXPIRE, MODIFY, and SUBTRACT. Also, the maximum terminal time is set to zero for purposes of the line labeled TOT produced by STATUS PROJECT and the commands ADD and SUBTRACT.

Hours

The keyword TERMHRS is used to specify changes in hours of maximum terminal time permitted to a signon ID. To restrict S002 to 2 hours of terminal time, entering

```
MODIFY S002,TERMHRS=2
```

produces

```
 SIGNON  CHARGE   DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
 S002       5.00           0       2:00       0:00 07-07-70
           10.00           0       2:00       0:00   16:25
```

IGNORE may also be used as a value for the keyword TERMHRS.

Changing Maximum PLOTting Time

Minutes

The keyword PLOT is used to specify changes in minutes of maximum plotting time permitted to a signon ID. To give S003 an hour and 45 minutes of plotting time, entering

```
MODIFY S003,PLOT=105
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003      140.00          15   IGNORED        1:45 12-31-70
          125.00           2      0:00        0:00   24:00
```

If the project is permitted unlimited plotting time, subject only to available funds, the word IGNORE may be used as the value of the keyword PLOT. For example,

```
MODIFY S003,PLOT=IGNORE
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003      140.00          15   IGNORED    IGNORED  12-31-70
          125.00           2      0:00        0:00   24:00
```

See also the analogous discussion of unlimited terminal time above.

Hours

The keyword PLOTHRS is used to specify changes in hours of maximum plotting time permitted to a signon ID. To give S003 one hour of plotting time, entering

```
MODIFY S003,PLOTHRS=1
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003      140.00          15   IGNORED        1:00 12-31-70
          125.00           2      0:00        0:00   24:00
```

IGNORE may also be used as a value for the keyword PLOTHRS.

Changing EXPIRation Time

The keyword EXPIRE is used to specify changes in the expiration time of a signon ID. The value of EXPIRE may be either of two forms.

Date

The form MM-DD-YY
where MM is the two digit month,
DD is the two digit day of the month,
and YY is the two digit year,

is used to set the expiration time to 12:00 p.m. of the specified date. For example,

```
MODIFY S003,EXPIRE=12-07-70
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003      140.00          15   IGNORED        1:00 12-07-70
          125.00           2      0:00        0:00   24:00
```

Time

The form MM-DD-YY/hh:mm
where MM-DD-YY is explained above,
hh is the two digit hour in twenty-four hour notation,
and mm is the two digit minutes,

is used to set the expiration time to a particular minute of the specified date. For example,

```
MODIFY S002,EXPIRE=11-23-70/16:45
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S002        5.00           0      2:00        0:00 11-23-70
           10.00           0      2:00        0:00   16:45
```

and S002 will not be able to sign on after 4:45 p.m. on November 23, 1970. Note that the expiration date and time may not be set to a value earlier than the current date and time.

Note that leading zeros must be supplied for the two digit items when there is only one significant digit. Also, the keyword EXPIRE behaves as though it were used with MODIFY, even when used with ADD, EQULIZE, or SUBTRACT. That is,

```
ADD S002,CHARGE=5,EXPIRE=11-23-70/16:45
```

and

```
SUBTRACT S002,CHARGE=5,EXPIRE=11-23-70/16:45
```

have the same effect on the expiration time as the above example but the effect on the maximum amount of money is quite different in each case.

If an attempt is made to expire a signon ID before both the current time and the current expiration time, the earlier of these two times is used as the expiration time. Thus,

```
MODIFY S004,EXPIRE=09-01-69
```

produces

```
SIGNON  CHARGE   DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S004      90.00           0       0:00       0:00 07-07-70
            0.00          0       0:00       0:00   16:25
```

when the current time is 4:25 p.m. on July 7, 1970.

If a signon ID is expired, no charge is made for file space belonging to that signon ID after the expiration time. In return, the right to destroy the files belonging to expired signon ID's is reserved by the Computing Center. However, the file space is still considered to be part of the cumulative file space for the project until it is actually destroyed. If the expiration time of the signon ID is set to a later time before the file space is destroyed, the project is charged for all file space used during the previously expired time.

The fact that a signon ID is expired has no effect on the changing of maximums by PROJECTACCOUNT. The expiration date and time may not be changed for the master signon ID.

Changing NOCHANGE

When using a block or ENTIRE for the signon range with any command except STATUS, certain signon ID's within the range may not be changed. If a signon ID has "no change" on, it is ignored by the command and no listing line is produced for it. The keyword NOCHANGE is used to specify the status of "no change". The words ON or OFF may be used as the value of the keyword. To have S003 ignored by commands that produce changes, entering

```
MODIFY S003,NOCHANGE=ON
```

produces

```
SIGNON  CHARGE   DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003      140.00          15    IGNORED       1:00 12-07-70
  NC      125.00           2       0:00       0:00   24:00
```

Note that the letters NC under the signon ID indicate that "no change" is on. Now entering

```
EQUALIZE ENTIRE,DISK=2
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S002       5.00           2       2:00       0:00 11-23-70
          10.00           0        2:00       0:00   16:45

S004      90.00           2       0:00       0:00 07-07-70
           0.00           0       0:00       0:00   16:25
```

and entering

```
STATUS S003
```

which produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003     140.00          15   IGNORED        1:00 12-07-70
  NC     125.00           2       0:00       0:00   24:00
```

verifies that S003 was not changed.

To change S003,

```
MODIFY S003,NOCHANGE=OFF
```

turns off "no change" and produces

```
SIGNON  CHARGE    DISK SPACE   TERM TIME  PLOT TIME  EXPIRE
S003     140.00          15   IGNOFED        1:00 12-07-70
          125.00           2       0:00       0:00   24:00
```

To change the maximums for a signon ID which has "no change" on and leave "no change" on at the end of the operation, the keyword parameters NOCHANGE=ON and NOCHANGE=OFF should both appear. For example,

```
ADD S003, NOCHANGE=ON, NOCHANGE=OFF,DISK=2
```

produces

```
SIGNON  CHARGE    DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S003     140.00          17   IGNORED        1:00 12-07-70
  NC     125.00           2       0:00       0:00   24:00
```

Note that the order of the keyword parameters makes no difference.

NOCHANGE behaves the same with any command except STATUS, which ignores it. It can be used to inhibit the changing of a signon ID which is not currently being used. A recommended method of expiring a signon ID, for example S004, is to first destroy all files belonging to the signon ID, and then enter

```
MODIFY S004,CHARGE=0,DISK=0,TERMINAL=0, EXPIRE=03-01-00,NOCHANGE=ON
```

to produce

```
SIGNON  CHARGE   DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
S004       0.00           0       0:00       0:00 07-07-70
  NC       0.00           0       0:00       0:00   16:25
```

Note that this is the same as entering

```
EXPIRE S004
```

after destroying the files.

Producing NOLISTing

To not print a listing line for each signon ID, the inclusion of the keyword NOLIST suppresses printing for that command only. NOLIST does not require a value. For example,

```
ADD S002,CHARGE=15,NOLIST
```

Producing a HEADING

To produce a heading at the top of a new page before printing the next listing line, the keyword HEADING should be included with the command. For example,

```
STATUS S002,HEADING
```

produces

```
SIGNON  CHARGE   DISK SPACE  TERM TIME  PLOT TIME  EXPIRE
      (DOLLARS)    (PAGES)   (HRS:MIN)  (HRS:MIN) DATE/TIME
(MAXIMUM AMOUNT IS ABOVE USED OR CURRENT AMOUNT.  NC=NO CHANGE ON.)

S002       20.00          2       2:00       0:00 11-23-70
           10.00          0       2:00       0:00   16:45
```

HEADING and NOLIST are the only keywords which do not require a value.

MISCELLANEOUS

A signon ID cannot be changed while it is signed on to the computer (with the exception of the master signon ID). If an attempt is made to change a signon ID while it is in use, an appropriate comment is printed and the change must be made at a later time.

The value of a maximum for a signon ID can always be reduced. However, if an increase in the value of a maximum would cause the cumulative for the project (second line produced by STATUS PROJECT) to exceed the maximum for the project, the increase is not permitted. Thus, it may be possible to subtract a given amount from a signon ID and impossible to add the same amount back because the cumulative for the project originally exceeded the project maximum.

If an attempt to change the maximums for a signon ID causes an error, the maximums are not changed, no listing is produced for the signon ID, and an appropriate error comment is printed.

An attention interrupt causes a question mark prefix to be printed indicating that the program is ready for the next input line. If a signon ID is being processed at the time of the interrupt, the processing is completed before the interrupt takes effect.

The used amount of money in the listing includes charges for file storage to the current time.

It is not necessary for a project maximum to be represented completely by the corresponding signon ID maximums, that is, the project maximum may be greater than the sum of the corresponding signon ID maximums. If this is the case, to increase the maximum for a signon ID, merely enter the appropriate input line (with the provision that the new maximum does not cause the cumulative for the project to exceed the project maximum).

The project cumulative for file space is computed as described in the section "PROJECT" with one exception for compatability with MTS. Before computing the maximum of the maximum and current space for each signon ID, the value to be used for current space is reduced by the smaller of 16 pages and the truncated integer result of dividing the maximum space by 8. For example, if the maximum is 7 pages, the value of the current space is used; if the maximum is 8 pages, the value to be used for current space is reduced by 1 page; and if the maximum is greater than or equal to 128 pages, the value to be used for current space is reduced by 16 pages.

ABBREVIATIONS

Both commands and keywords may be abbreviated. Only the first three letters of the command are necessary; minimum abbreviations for commands and keywords are underlined in the summary below.

Thus, the last example in "Changing NOCHANGE" can be expressed more concisely as

```
MOD S004,C=0,D=0,T=0,P=0,E=03-01-00,NC=ON
```

SUMMARY OF INPUT FOR PROJECTACCOUNT

Input Lines

The form of the input line, using standard notation, is

```
                 ┌          ┐   ┌          ┐
[b]... < STATUS > d < signon-range > [d HEADING] [d] [; comment]
       |        |   | PROJECT  |
       └        ┘   └          ┘
```

or

```
          ┌          ┐
          | ADD      |
 [b]... < | EQUALIZE | > d signon-range keyword... [d] [; comment]
          | MODIFY   |
          | SUBTRACT |
          └          ┘
```

or

```
          ┌          ┐
[b]... < CONTINUE > d signon-range [keyword]... [d] [; comment]
       | EXPIRE   |
       └          ┘
```

or

```
[b]... {PASSWORD} d signon [d] [; comment]
```

Where d is {b|,}...

keyword is

```
                  ┌ ┌          ┐          ┐
                  | | CHARGE   |          |
                  | | DISK|FILE|          |
                  | | EXPIRE   |          |
                  | < NOCHANGE > =value   |
d < | PLOT       |          >
                  | | PLOTHRS  |          |
                  | | TERMHRS  |          |
                  | | TERMINAL |          |
                  | └          ┘          |
                  | HEADING               |
                  | NOLIST                |
                  | NOHEAD                |
                  └                       ┘
```

```
                          ┌                          ┐
                          | signon                   |
signon-range is           < signon d... [d] signon   >
                          | ENTIRE                   |
                          └                          ┘
```

b	is a blank or space,
d...	is a delimiter followed by three dots (periods),
{}	indicate a compulsory choice of one line,
|	indicates a compulsory choice of one element,
[ ]	indicate an optional choice,
...	(not d...) indicates an optional (0 or greater) number of repetitions of the preceding element,
lower case letters	indicate a type of element to be used,
and all other characters	are to be used exactly as shown.

Comments may appear at the end of the input line and must be separated from the last parameter by a semi-colon.

Commands

Command	Function
ADD	Adds the values of the keyword parameters to the current maximums to produce the new maximums.
CONTINUE	Produces the same action as the previously entered command but uses the new signon range and any new keyword values explicitly specified.
EQUALIZE	Adds the values of the keyword parameters to the used or current amounts to produce the new maximums.
EXPIRE	Substitutes zero for the current maximums and sets the expiration time to the current time unless other keyword values are explicitly specified, and sets "no change" on.
MODIFY	Substitutes the values of the keyword parameters for the current maximums.
PASSWORD	Enables project director to display current password for any signon ID belonging to the project.

STATUS	Lists the current values of maximum and used or current amounts.
SUBTRACT	Subtracts the values of the keyword parameters from the current maximums to produce the new maximums.

Keywords

Keyword	Reference	Value Units
CHARGE	Money	Dollars
[DISK|FILE]	Disk Space	Pages
EXPIRE	Expiration Time	MM-DD-YY MM-DD-YY/hh:mm
HEADING	Listing Heading	none
NOHEAD	Listing Heading	none
NOCHANGE	No Change	ON OFF
NOLIST	Listing	none
PLOT	Plotting Time	Minutes IGNORE
PLOTHRS	Plotting Time	Hours IGNORE
TERMHRS	Terminal Time	Hours IGNORE
TERMINAL	Terminal Time	Minutes IGNORE

Termination

An end-of-file on input terminates execution.

Logical I/O Unit Names

SCARDS - Input
SPRINT - Listing
SERCOM - Error Comments

April 1974

Batch Input to Produce Examples in this Section

```
$SIGNON S001 '  PROJECT DIRECTOR'
PASWRD
$RUN *PROJECTACCOUNT
ADD S003,CHARGE=100
SUBTRACT S003,CHARGE=10
MODIFY S002,CHARGE=15
EQUALIZE S003,CHARGE=25
EXPIRE S002
EXPIRE S004,CHARGE=95
STATUS S001
ADD S002,CHARGE=5,NOCHANGE=OFF
CONTINUE S004
CONTINUE S002,CHARGE=10
PASSWORD S004
STATUS S002,...,S004
MODIFY S001,NOCHANGE=ON
SUBTRACT ENTIRE,CHARGE=10
STATUS PROJECT
ADD S003,DISK=5
MODIFY S002,TERMINAL=150
MODIFY S002,TERMINAL=IGNORE
MODIFY S002,TERMHRS=2
MODIFY S003,PLOT=105
MODIFY S003,PLOT=IGNORE
MODIFY S003,PLOTHRS=1
MODIFY S003,EXPIRE=12-07-70
MODIFY S002,EXPIRE=11-23-70/16:45
MODIFY S004,EXPIRE=09-01-69
MODIFY S003,NOCHANGE=ON
EQUALIZE ENTIRE,DISK=2
STATUS S003
MODIFY S003,NOCHANGE=OFF
ADD S003,NOCHANGE=ON,NOCHANGE=OFF,DISK=2
MODIFY S004,C=0,D=0,T=0,P=0,E=03-01-00,NC=0
ADD S002,CHARGE=15,NOLIST
STATUS S002,HEADING
$ENDFILE
```

April 1974

April 1974

GLOSSARY OF COMPUTING TERMS

This glossary is meant to be used as an aid in interpreting the computing terms that are used in the documentation produced by the Computing Center. This section attempts to define the normal use of each term and favors the "everyday" interpretations over the more technical descriptions.

All terms given in the first line of each entry, and separated by commas, are considered synonyms.

Each entry is given a code which is meant to give some indication of the scope of the definition. The codes have the following meanings:

GC — The term and definition is widely used in the computing field (GC = General Computing).

360 — The term and definition is valid in the context of the IBM 360 (and probably 370) series of computers, but not necessarily anywhere else.

MTS — The term and definition is valid in the context of the MTS operating system.

WSU — The term and definition is applicable only to the Wayne State University Computing and Data Processing Center.

Abend, Abnormal End — 360

An erroneous or incorrect program termination, indicating programming or data problems.

Address — GC

An identification, as represented by a name, label, or number, for a register or location in storage.

AFD — MTS

A pseudo device name referring to the currently active file or device. It is the file or device obtained by a GET, CREATE, or EDIT command, to which MTS writes (or from which it reads) data lines when in MTS command mode.

Algorithm — GC

A prescibed set of well-defined rules or processes for the solution of a problem in a finite number of steps; for example, a full statement of an arithmetic procedure for evaluating "sin x" to a stated precision.

Alphanumeric, Alphameric GC

Pertaining to a character set that contains both letters and digits.

American Standard Code for Information Interchange, ASCII GC

A character transmission code for use between a terminal device and a computer. Each character is uniquely represented by a 7-bit number with an eighth bit generated for parity. The parity bit is used as a check bit.

APL GC

A Programming Language; a conversational programming language which is best suited for problems requiring extensive manipulation of a small amount of data.

Application Package GC

A set of programs and/or sub-programs designed to aid in the solution of a specific class of problems; for example, the IBM package SSP (Scientific Subprogram Package) is designed to provide support for numerical and statistical programs.

Argument GC

A parameter explicitly passed from a calling program to a called program (subroutine); for example, in the FORTRAN statement CALL SUB(A,B,C), the variables A, B, and C are arguments to be passed to the subroutine SUB.

Array GC

An arrangement of elements in one or more dimensions.

ASCII - See American Standard Code for Information Interchange. GC

Assemble GC

To prepare a machine language program from a low-level symbolic language (assembler language) program by substituting absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

Assembler GC

A program that converts low-level symbolic language programs into machine code.

360-assembler Language 360

A low-level symbolic programming language used to write programs for the IBM 360 series of computers. See also Assemble and Assembler.

Auxiliary Storage, Secondary Storage GC

Data storage other than main storage; for example, storage on magnetic tape or direct access devices. In general, any storage that supplements another storage is auxiliary. See also main storage.

Backspace GC

To move backward one or more characters or records; for example, if an input device (such as magnetic tape) is positioned to read the nth record, a backspace of one record positions the device to read record n-1.

Base Address 360

An address used as a base to compute a storage location. Instructions which refer to a storage location do so by specifying a register and a 12-bit displacement address. The register contains a 24-bit (or 32-bit) base address. The actual storage location referenced is given by the sum of the base address and the displacement. See also Base Register.

Base Register 360

Any general register used to hold a base address. See also Base Address.

Batch Processing GC

A form of computer job processing in which the jobs (usually in the form of card decks or card images) are submitted to a central location where they are queued, either manually or automatically, for execution at a later time. There is no possibility of interacting with a batch job. See also HASP.

Batch Queue, Batch Stream GC

In batch processing, the queue in which computer jobs await execution. See also Batch Processing, HASP.

Binary GC

(1) Pertaining to the number system with the base of two.
(2) Pertaining to a characteristic or property involving a selection, choice, or condition in which there are only two possibilities.

Binary Code GC

A code that makes use of exactly two distinct characters, usually 0 and 1.

Binary Coded Decimal, BCD GC

A character code which represents each character uniquely in 6 binary digits. Hence, there are 64 characters possible within the encoding.

Binary Digit, Bit GC

One of the digits (0 or 1) of the binary number system. This term is usually contracted to "bit" in computer terminology.

Binary Format GC

A term referring to "unformatted" I/O in FORTRAN programs; for example, the format of the output from the FORTRAN statement WRITE (4) X,Y,Z. "Unformatted" is the preferred terminology.

Binary Read Feature 360

A feature on a card reader that allows the reading of column binary cards, that is, cards with non-EBCDIC punches within any or all columns.

Binary Search GC

An efficient method of searching an ordered table, in which the midpoint of the table is examined and one half is rejected as not containing the elements desired. The process is repeated on the half containing the elements. Successive bisection of the table quickly isolates the desired elements. This method is also known as the half-interval search.

Bit - See Binary Digit. GC

Block - See Physical Record. GC

Blocking GC

The process of combining two or more logical records into one physical unit (physical record) so that it can be written to or read from an I/O device in a single operation.

Blocking Factor GC

The number of logical records combined to make one physical record. See also Blocking.

BPI (bits per inch) - See Magnetic Tape Density. GC

Buffer GC

A temporary storage region for data being transmitted from one unit to another; for example, from a central processor to an I/O unit. The purpose of the buffer is to compensate for the different speeds at

which the units can handle data. Sometimes a buffer is a permanent hardware feature of the unit (for example, as in a "buffered printer") and sometimes internal main storage areas are assigned temporarily to act as buffers.

Bug GC

A mistake in a program which causes a malfunction.

Byte 360

A sequence of 8 adjacent binary digits (bits) which can be addressed and operated on as a unit. A byte is the smallest addressable unit of storage in the IBM 360 series of computers.

CALCOMP Plot GC

A diagram produced by a CALCOMP plotter. See also CALCOMP Plotter, Digital Incremental Plotter, Plotter.

CALCOMP Plotter GC

A plotter manufactured by California Computer Products, Inc. See also Digital Incremental Plotter, Plotter.

Called Program GC

The routine to which a calling program transfers control. See also Calling Program.

Calling Program GC

The routine from which a routine has been called. See also Called Program.

Calling Sequence GC

A specified arrangement of instructions and data necessary to transfer information and control to a routine.

Card, Computer Card, Punched Card GC

A card which can be used as a computer input or output medium. There are various sizes and shapes of cards, the most common having 80 columns and measuring 7 3/8 x 3 1/4 inches.

Card Column GC

The most common computer card has 80 vertical columns numbered from 1 to 80 from left to right. Each column is divided into 12 punching positions and can be used to contain the coded representation of an alphanumeric character. The code is a series of punched holes.

Card Deck, Deck GC

A collection of punched cards comprising programs and/or data for a computer run.

Card Hopper, Hopper GC

A holder into which cards are placed, and from which they are fed to the punching or reading mechanism of a card punch or reader.

Card Image GC

A one-to-one representation of the contents of a computer card; for example, a magnetic tape containing one-to-one copies of the cards in a card deck is said to contain card images.

Card Punch GC

A peripheral unit for punching computer cards. See also Keypunch.

Card Reader GC

A peripheral unit for reading computer cards.

Card Stacker GC

A holder into which computer cards are accumulated into a deck after they have been either read or punched.

Carriage Control Character GC

A character, normally the first in an output record, which is used to control the vertical spacing of printed output and is not itself printed. See also Logical Carriage Control, Machine Carriage Control.

Cathode Ray Tube, CRT GC

An electronic tube in which a beam of electrons can be controlled and directed by an electronic lens so as to produce a visible display of information on the surface of the tube. The IBM 2260 Display Station terminal and the Tektronix 4010 Storage Tube terminal use CRT's.

Ccid MTS

A 4-character identification code assigned to each user of the MTS system. The ccid serves to identify the user's account. It is also referred to as a "signon id".

Central Processing Unit, CPU, Central Processor GC

The unit in any digital computer system which coordinates and controls the activities of all the other units and performs all the logical and arithmetic processes to be applied to data.

Character GC

A letter, digit, or other symbol that is used as part of the representation of data.

Character Code GC

A numeric code assigned to characters so that they may be represented in the memory of digital computers. See also Binary Coded Decimal, Collating Sequence, Extended Binary Coded Decimal Interchange Code.

Character String GC

A sequence of characters.

Checkpoint GC

A place in a program where a check, or a recording of data for restart purposes, is made.

Closing a File MTS

The process of causing all main storage buffers containing parts of the file that were changed to be written to the file, and then causing all buffers and control blocks associated with the file to be released.

COBOL GC

Common Business Oriented Language; a procedural language designed for commercial data processing.

Code GC

(1) A set of unambiguous rules specifying the way in which data may be represented.
(2) To write a program.

Collate GC

To compare and merge two or more similarly ordered sets of items into one ordered set.

Collating Sequence GC

An ordering assigned to a set of characters, the ordering being determined by a numeric code assigned to each character. See also Character Code.

Collator GC

A machine used to merge two independent card decks which have been sorted into collating sequence.

Column - See Card Column. GC

Column Binary GC

Pertaining to the binary representation of data on computer cards in which adjacent positions in a column correspond to adjacent binary digits. Loosely, and more understandably, any non-standard, that is, non-EBCDIC, card punch sequences. See also Binary Read Feature.

Command Language GC

The set of commands which interface the computer user with the computer system, for example, the MTS commands.

Command Line MTS

An input record containing a command, such as an MTS command, debug command, edit command, etc.

Command Mode - See MTS Command Mode. MTS

Common Storage Area GC

Usually, main storage which can be shared by several routines.

Compilation Time GC

(1) The length of time it takes to translate a program from the source language into machine language.
(2) The time during which a program is being compiled.

Compile GC

To translate a program written in a high level language (for example, FORTRAN, ALGOL, PL/I) into an equivalent machine language program which may subsequently be executed.

Compiler GC

A program which converts a high level language (for example, FORTRAN, ALGOL, PL/I) into a lower level language, usually machine language. A compiler normally generates several low level instructions for each source language statement. In addition, most compilers allow the programmer to incorporate sub-programs from a program library.

Completion Code GC

A return code from a language processor which indicates whether any serious errors were encountered and, if so, how serious they were. See also Return Code.

Computer Card - See Card. GC

Computer Graphics, Graphics GC

Any form of drawing, (for example, maps, graphs) produced under computer control. The term is sometimes generalized to include any form of visible computer output, thereby including printed output as well as display on CRT screens. See also Calcomp Plotter, Digital Incremental Plotter, Plotter.

Computer Word - See Fullword. GC

Concatenate MTS

To connect two or more sets of information, such as strings or files, in sequence, such that they can be treated as a single unit. See also Explicit Concatenation, Implicit Concatenation.

Console - See Operator's Console. GC

Console Typewriter GC

A unit of the operator's console through which communication between the operator and the operating system takes place. Console typewriter messages include those informing the operator of some required action, for example, tapes requiring mounting, and those informing the operator about the status of the programs currently running. See also Operator's Console.

Continuous Systems Modeling Program - See CSMP. GC

Control Card GC

A card or statement in a job which contains a command to the operating system or to a program.

Control Character GC

A character whose occurrence in a particular context initiates, modifies, or stops a control operation; for example, a character to control spacing on a printer.

Control Program GC

A program which provides such functions as the handling of I/O operations, error detection and recovery, program loading, and communication between the program and the operator.

Conversational, Interactive GC

Describes the interaction or dialogue which takes place (via a terminal device) between a user and a program running under a time-sharing system (such as MTS).

Conversational Mode, Interactive Mode GC

A method of computer operation in which the user is in direct communication with the operating system and is able to control, interrogate, modify, and observe the processing of his task.

Conversational Terminal, Interactive Terminal GC

Normally a typewriter-like or CRT display device from which a user can communicate with a time-sharing system (such as MTS) on a real-time basis.

Convertor - See Data Convertor. GC

CPI (Characters per inch) - See Magnetic Tape Density. GC

CPU - See Central Processing Unit. GC

CPU Bound GC

Describes a job which has a high computation content in relation to both the number of service requests (for example, I/O requests) and the total real time it spends on the computer. See also I/O Bound.

CPU Time GC

The time taken by the central processing unit to perform the operations requested.

Cross-reference Listing GC

An alphabetic list of all symbols used in a program and an ordered list of the numbers of all statements which reference a given symbol. The symbols can be statement labels, variable names, and so on.

CRT - See Cathode Ray Tube. GC

CSMP GC

Continuous Systems Modeling Program; a program package which allows the simulation of continuous processes.

Cycle Time GC

The length of time a computer takes to perform its most primitive operation. The cycle time is a measure of the basic speed of a computer.

Data Line MTS

Any line, not beginning with a "$", entered when the system is in MTS command mode, which cannot be interpreted as a command, and which has a line number associated with it. Data lines are written to the currently active file or device, if there is one.

Data Record GC

A record, for example a punched card, containing data to be processed by a program. See also Record.

Data Set GC

(1) A collection of data in one of several arrangements and described by control information to which the system has access; for example, in MTS, a file is a data set.
(2) A device which performs the modulation/demodulation and control functions necessary to provide communication between computers and related devices via telephone lines.

Debug GC

To detect, locate, and remove mistakes from a program, or malfunctions from equipment.

Debug Mode MTS

The state in which the Symbolic Debugging System (SDS) is in control and monitoring a program which is being debugged. See also Symbolic Debugging System.

Deck - See Card Deck. GC

Default Definition, Default Option GC

An assumption which is made about the value of a variable or parameter if no explicit choice is given by the user.

Default I/O Unit Assignments MTS

Assumptions which are made about the assignment of I/O units if no explicit assignments are given; for example, SCARDS defaults to *SOURCE* if it is not explicitly assigned to another file or device.

Default Option - See Default Definition. GC

Delimiter, Separator GC

A special indicator that separates and organizes items of data; for example, the quote marks (') enclosing literal character data in FORTRAN, the blanks that separate the elements of an MTS command, a $ENDFILE card at the end of a data deck.

Delivery Code WSU

The first character within single quotes on the signon card on a batch job indicating where the printed and punched output should be manually delivered by the campus delivery system.

Density, Record Density GC

The number of bits in a single linear track measured per unit of length of the recording medium. See also Magnetic Tape Density.

Device, Unit GC

Input-output peripheral equipment; for example, a card reader, magnetic tape drive, terminal, etc.

Device Command MTS

A command, given in conversational mode, intended for the device support routine of the terminal at which it is given. These usually invoke (or revoke) special editing capabilities available at the terminal and should not be confused with MTS commands. The device support routines interpret device commands and never pass them on to MTS. See also Device Support Routine.

Device Name MTS

A unique four character name assigned to each device. The device name is used to refer directly to a particular device. For example, the device name assigned to a particular magnetic tape drive is T0C2. Device names do not normally concern the average programmer, since he is more likely interested in classes of devices, rather than individual devices. Thus, he most often uses pseudo-device names, instead of device names. See also Device, Pseudo-device Name.

Device Support Routine, DSR MTS

A subroutine (or set of subroutines) which interfaces a particular device to the MTS operating system. The DSR handles all actual I/O with the device, performing error recovery if necessary, recognizing attention interrupts, etc. Conversational terminal device support routines also provide several editing and typographical conveniences which are enabled or disabled via device commands. See also Device, Device Command.

Diagnostic - See Error Message. GC

Digital Computer GC

A machine capable of performing arithmetic and logical operations on data represented in numeric or character form.

Digital Incremental Plotter GC

A device that employs digital signals from a central processing unit to activate a plotting pen and a drum which carries the plotting paper. The plotting action results from step motions, that is, incremental motions, of the pen and/or drum. See also Calcomp Plotter, Plotter.

Direct Access, Random Access GC

Pertaining to the process of obtaining data from, or placing data into, main storage, where the time required for such access is relatively independent of the location of the data most recently referenced. For example, MTS line files allow direct access input and output.

Direct Access Device GC

A device in which the access time is effectively independent of the location of the data being accessed, for example, disks are direct access devices.

Direct Access File GC

A file in which a given record may be accessed directly, without regard to other records in the file or to the relative location of the record in the file. MTS line files are direct access files.

Disc - See Disk. GC

Disk, Magnetic Disk GC

A storage device consisting of a number of rotating flat circular plates, each coated on both surfaces with some magnetizable material. A number of concentric circular tracks are available on each surface. Data is read from or written to these tracks by means of read/write heads that can move radially to access a track. There may be several heads to each surface, a particular head being allocated a specific area (or sector) on the disk. In MTS, disks are used for line and sequential file storage, HASP batch queue storage, and paging.

Dismounting - See Magnetic Tape Dismounting. GC

Displacement 360

All machine instructions which designate a main storage location contain a 12-bit number, called the displacement, which provides for relative addressing of up to 4095 bytes beyond the base address. The base address and displacement are added together to produce the actual address referenced.

Display Unit GC

A device that provides a visual display of data on a CRT screen. IBM 2260 terminals and Westinghouse 1600 terminals are examples of such devices.

Double Precision GC

A number representation in which two consecutive computer words are used to hold a single numerical value. On some computers this effectively doubles the precision of numerical calculations. With the IBM 360 series, double precision calculations involve approximately 16 decimal digits of precision. See also Single Precision.

Doubleword 360

A sequence of two fullwords (8 bytes) capable of being treated as a unit. Such a sequence must have an address that is divisible by 8. Double precision floating point values are stored in doublewords.

Drive - See Magnetic Tape Drive. GC

Drum, Magnetic Drum GC

A right circular cylinder with a magnetic surface on which data can be stored by selective magnetization of circular tracks on the curved surface. Data access is by means of stationary read/write heads positioned over each track of the rotating magnetic surface. In MTS, drums are used for paging and are not directly accessible to users of the system.

DSR - See Device Support Routine. MTS

DUMMY MTS

A special system pseudo-device which is always empty. On output, it represents an infinite waste basket; lines are accepted and disappear. On input, *DUMMY* acts like an empty file; every time a line is requested, an end-of-file condition is given.

Dummy Parameter, Dummy Variable GC

A parameter used in the definition of a sub-program. These are effectively replaced by the actual parameters provided in a call to the sub-program.

Dummy Routine GC

A sub-program that (temporarily) replaces a routine that is missing, either because it is no longer necessary, or because it has yet to be written. Dummy routines are often provided during the debugging stage of program development.

Dummy Variable - See Dummy Parameter. GC

Dump GC

To copy all or part of a storage device (main storage, magnetic tape) to another device (a printer) for subsequent examination. The term is also used to refer to the process by which the copy is made, and the actual results (printed output) of the copy.

Duplex System GC

A system in which two distinct central processing units operate in tandem to handle the work load. That is, both CPU's have access to the same facilities and job queues. A program may be executed by either or both CPU's. The WSU IBM System/360 Model 67 is a duplex system.

EBCDIC - See Extended Binary Coded Decimal Interchange Code.

Edit GC

To modify the form or content of data; for example, to insert or delete characters such as page numbers or decimal points.

Edit Mode MTS

The state in which the context editor is in control for the purpose of making changes to a file.

Editor GC

A program which performs editing functions.

Elapsed Time GC

The total apparent real time taken by a calculation, as measured by the time between the apparent beginning and the apparent end of the calculation. This is normally longer than the CPU time devoted to the calculation since active processing does not necessarily go on at all times.

End-of-file, EOF GC

The indication of the end of a data set. Various end-of-file indicators (depending upon the device on which the data resides) are used to mark this point. There are hardware end-of-file indicators which always mark the physical end of data; for example, a magnetic tape file mark, or the last card read after the "endfile" button is pushed on a card reader. In addition, there are software end-of-file indicators which usually mark a logical, rather than physical, end of data; for example, in MTS, the characters $ENDFILE in columns 1 to 8 of a card. See also Magnetic Tape File Mark.

End-of-file Mark - See Magnetic Tape File Mark. GC

End-of-reel Marker - See End-of-tape Marker. GC

End-of-tape Marker, End-of-reel Marker GC

A strip of reflective material which signals the end of the usable portion of a magnetic tape. This strip can be automatically detected by a tape drive.

Entry Point GC

In a program, any point to which control can be passed from an external program.

EOF - See End-of-file. GC

Error Message, Diagnostic GC

An indication, usually a printed message, that an error has been detected.

Executable Statement GC

A statement in a programming language which, when processed (or compiled), causes executable object code to be generated.

Execute GC

To carry out an instruction or perform (run) a routine.

Execution Time, Object Time, Run Time GC

(1) The length of time it takes to execute a program.
(2) The time at which a program is being executed.

Explicit Concatenation MTS

A form of concatenation in which files or devices are chained together to appear like a single file or device, for example, the explicit assignment of a single logical I/O unit to chained files or devices. The files or devices are chained by giving the appropriate file, device, or pseudo-device names (with line number ranges or modifiers, if desired) separated by "+" signs. For example, 5=FILEA(1,23)+* SOURCE* assigns logical I/O unit 5 to an explicit concatenation of the lines of FILEA with numbers between 1 and 23, and the *SOURCE* pseudo-device. See also Concatenation, Implicit Concatenation.

Extended Binary Coded Decimal Interchange Code, EBCDIC 360

The character code used by IBM 360 computers to represent characters. Each character is uniquely represented by an 8-bit number, thus giving 256 possible characters of which approximately 125 have been assigned

commonly used graphic and control functions. In addition to the numeric code, there is also an EBCDIC card punch code, which represents each character as a pattern of punched holes on a computer card.

Extent - See File Extent. MTS

External Symbol Resolution GC

The process of linking together, during the loading process, all references of one (sub) program to any others.

FDname MTS

An abbreviation for file or device name.

Fetch Protection GC

A feature which prevents the accessing of data from specified areas of main storage. See also Memory Protect.

Field GC

In a record, a specified area used for a particular category of data; for example; a group of card columns used to represent a wage rate.

File GC

In general, a collection of related information, usually organized into smaller units called records. With MTS, the term is usually used to refer to files residing on direct access devices, although it is often used in reference to data on tapes. See also Line File, Private File, Public File, Sequential File, Sequential File With Line Numbers, Temporary File.

File Extent, Extent MTS

A block of contiguous disk tracks allocated to an MTS line or sequential file.

Filemark - See Magnetic Tape File Mark. GC

File Protection GC

The prevention of accidental overwriting of data files. File protection can be by hardware (file protect rings on magnetic tapes) or software (by causing a program to check file names).

File Protect Ring, Ring, Tape Ring, Write Permit Ring GC

A ring of plastic material which must be inserted into the slot provided on the back of a magnetic tape reel in order to allow writing of data on the tape. The presence or absence of a ring is detected by the tape drive hardware. (No ring, no write.)

Fixed Point GC

Pertaining to a numeration system in which the position of the decimal point is fixed with respect to one end of the numerals, according to some convention. For example, FORTRAN INTEGER constants and variables are fixed-point numbers with the decimal point assumed to be at the extreme right of all of the digits.

Flag GC

Anything (for example, a variable in a program) that is used to signal the occurrence of some condition.

Floating Point GC

Pertaining to a numeration system in which the position of the decimal point does not remain fixed with respect to one end of the numerals. This notation usually provides for some representation of a power of the base, to act as a scale factor. For example, FORTRAN REAL constants and variables are floating-point numbers.

Flowchart GC

A graphical representation of the definition, analysis, or solution of a problem, in which symbols are used to represent operations, data flow, decisions, equipment, etc.

Format GC

A specific arrangement of data.

FORTRAN GC

An acronym for Formula Translator. FORTRAN is a problem oriented high level programming language generally suited for scientific and mathematical use.

FORTRAN-callable GC

Used in reference to a routine which is capable of being invoked directly by a FORTRAN program.

FORTRAN G Compiler GC

The IBM Level G FORTRAN IV compiler. It is used to compile FORTRAN IV source programs and to perform other auxiliary services. See also FORTRAN H Compiler, WATFOR.

FORTRAN H Compiler GC

The IBM Level H FORTRAN IV compiler. It serves the same purpose as the FORTRAN G compiler (namely, the compilation of FORTRAN IV programs) but is slower and more expensive to run. However, it produces object decks

which generally take up less computer storage and execute more quickly. FORTRAN H should only be used for completely debugged programs. See also FORTRAN G Compiler, WATFOR.

Fullword 360

A sequence of 4 bytes (32 bits) that can be treated as a unit in numeric and character machine operations. The address of the 4-byte sequence must be divisible by 4.

Global Time Limit MTS

The maximum CPU time allowed an MTS batch job from signon to signoff. The global time limit is given, either directly or by default, with the SIGNON command. See also Local Time Limit.

GPSS 360

General Purpose Simulation System; a program package which allows the simulation of discrete systems.

Graphics - See Computer Graphics. GC

GUSER MTS

An acronym for "get from user." GUSER is a logical I/O unit and defaults to *MSOURCE*. A user's responses to system and program queries are normally read from GUSER.

Halfword 360

A contiguous sequence of two bytes (16 bits) which comprises half of a fullword and is capable of being addressed as a unit. A halfword must have an address divisible by 2.

Halfword Integer 360

A number between -32768 and 32767 inclusive which may be stored within a halfword. In FORTRAN, a halfword integer is designated as INTEGER*2.

Hardware GC

Physical equipment, as opposed to a program (software) or a method of use; for example, card readers, line printers, card punches, magnetic tape drives, central processing units, input/output channels, etc.

HASP, HASP Batch Monitor, HASP Monitor 360

Houston Automatic Spooling and Priority System; a program which allows the queueing of batch input or output and a priority scheme for execution of batch jobs. See also Batch Processing, HASP Batch Mode, HASP Priorities.

HASP Batch Mode 360

Jobs submitted via HASP are sometimes said to run under HASP batch mode. See also HASP, HASP Priorities, Conversational Mode, Remote Batch Mode.

HASP Batch Monitor - See HASP. 360

HASP Priority 360

A parameter assigned to a HASP batch job, either by HASP itself or by the user, which determines, in part, the order in which the job is processed.

Hexadecimal, Hex GC

(1) Pertaining to the number system with a base of 16.
(2) Pertaining to a characteristic or property involving a selection, choice, or condition in which there are 16 possibilities.

High-level GC

A term describing programming languages that require extensive translation before they can be understood by computers. PL/I, FORTRAN, ALGOL are examples of such languages.

Hollerith GC

A term often used as a synonym for "character". See also Hollerith Code.

Hollerith Code GC

A punched card character code invented by Dr. Herman Hollerith in 1888.

Hopper - See Card Hopper. GC

ID - See Ccid. MTS

Image - See Card Image. GC

Immediate Value GC

A symbol which is considered to be a value, rather than a reference to a value.

Implicit Concatenation MTS

A method of linking input files or devices together, so that they appear to be a single file or device, through the use of a "$CONTINUE WITH" line within each of the files or devices to be linked (except the last). See also Concatenation, Explicit Concatenation.

Indexed GC

Describes a method of data organization in which any record can be accessed directly by means of an index of record numbers. MTS line files are indexed according to MTS line numbers.

Initial Program Load, IPL GC

The initialization procedure that loads the operating system into the computer. The procedure must be carried out when the computer is started up, and when software or hardware problems necessitate a system restart.

Input GC

Information or data transferred or to be transferred from an auxiliary storage medium into the internal main storage of a computer.

Input Device GC

A device from which data can be read into the computer.

Input File GC

A file from which data is read.

Input/Output, I/O GC

A term used to describe anything (such as, equipment, data, programs, and so on) involved in communication with a computer.

Input/Output Device GC

Any part of the computing system hardware which is primarily used for the transmission of information from one type of storage to another.

Input Record GC

A record read into or to be read into main storage from an auxiliary storage medium.

Instruction GC

That part of a computer program which tells the computer what function to perform. An instruction consists of the contents of a storage location or series of storage locations subdivided into groups which represent coded commands to the computer. An operation code, such as the codes for add or subtract, may be specified, along with one or more addresses which specify the locations of operands to which the command is to apply.

Integer Variable GC

A variable which can take only integer values.

Interactive - See Conversational. GC

Interactive Computer Graphics GC

A form of computer processing in which a user can direct, in real time, the course of action of a program by interacting with a visual display produced by the program. See also Computer Graphics.

Interactive Terminal - See Conversational Terminal. GC

Interblock Gap - See Record Gap. GC

Internal Representation GC

This term refers to the manner in which numbers, characters, and instructions are stored within the computer's main storage. In most digital computers, the method of representation is based on the binary number system in which all elements are combinations of the binary digits, 0 and 1. The binary system is most widely used because of the convenience of constructing logic circuits and storage devices capable of handling data in this form.

Internal Statement Number, ISN 360

Numbers generated by compilers such as FORTRAN G and FORTRAN H and usually listed to the left of each statement. These numbers are generated so that every statement has a unique number which can be used in error comments and cross-reference listings to refer to the statement. These numbers should not be confused with MTS line numbers.

Interpret GC

(1) To print on a punched card the readable symbols corresponding to the punches.
(2) To analyze and immediately execute a source language program.

Interpreter GC

(1) A machine used to interpret punched cards.
(2) A program which executes source code as it encounters it, rather than after it has translated all the source code.

Inter-record Gap - See Record Gap. GC

Interrupt, Trap GC

An interrupt is essentially a subroutine call to a specific location in main storage invoked automatically by the hardware rather than by a program. It is used to inform the operating system of the occurrence

of some internal or external event that requires some action on the part of the operating system. For example, a card reader which has finished reading a card interrupts the CPU to inform it that the I/O operation is complete; a program that attempts to divide by zero interrupts the CPU so that diagnostic action can be taken, and so on. The operating system deals with interrupts according to a complex algorithm that depends upon the type of interrupt, the current state of the system, and so on.

Interrupt Code GC

A numeric code which serves to identify the type of an interrupt. In IBM 360 machines (except Model 67's in extended PSW mode), the interrupt code becomes part of the program status word when an interrupt occurs.

Interrupt Mask GC

A program controllable binary switch which, according to its value, prevents or allows an interrupt. There is one switch for each type of interrupt.

I/O - See Input/Output. GC

I/O Bound GC

Describes a job which issues a large number of input/output requests in relation to both the time required for computation and the total real time spent on the computer. See also CPU Bound.

I/O Modifier MTS

A character or set of characters beginning with an "@" appended to a file or device name in order to change the way in which it is treated or regarded during I/O processing.

IPL - See Initial Program Load. GC

Irrecoverable I/O Error GC

A persistent error condition on an input/output device such as a magnetic tape drive, disk, etc.

ISN - See Internal Statement Number. 360

JCL - See Job Control Language. 360

Job GC

A group of tasks prescribed as a unit of work for a computer. By extension, a job usually includes all necessary programs, data, and instructions to the operating system. In MTS, a job is delimited by the SIGNON-SIGNOFF commands.

Job Control Language, JCL 360

The language that allows the user to interact with the IBM Operating System/360 (OS/360) in order to use the computer.

Job Deck GC

A collection of punched cards comprising a complete batch job. On MTS at WSU, a job deck begins with an S-8 card, and is followed by a $SIGNON command, and some combination of commands, programs, and data, and terminates (optionally) with a $SIGNOFF command.

Justify GC

To align data about a specified reference. Frequently, the term is used in reference to data preparation. To "right justify" data is to move the data as far to the right in the allotted space (field) as possible. An analogous definition holds for "left justify".

K GC

In computer terminology, the symbol "K" usually represents two to the tenth power, that is, $1K=2^{10}=1024$. For some decimal computers, for example, the IBM 1401 and 1410, "K" represents ten to the third power, or $1K=10^3=1000$.

Key GC

One or more characters or numbers within an item of data which serves to identify or control the data or its use. For example in sorting, the "sorting key" is that number or string of characters in each data record according to which the data is sorted.

Keypunch GC

A keyboard operated machine used to punch holes into computer cards. The operator simply depresses keys on a typewriter-like keyboard to cause holes to be punched column by column into the card.

Keyword Parameter GC

A parameter in a command or programming statement which is of the form "KEYWORD=operand", for example, T=60 on the $SIGNON command, and MAP=*SINK* on the $RUN command. The operand is variable, the keyword name is fixed. In some cases, there is no operand and the form is "KEYWORD"; this is sometimes referred to as a degenerate keyword.

Label GC

One or more characters used to identify an item of data. The term is most often encountered in reference to a string of characters which serves to identify a statement in a program. It is also used in

reference to identifying information found on magnetic tape. See also Magnetic Tape Label.

Left Justify - See Justify. GC

Library File MTS

A file which contains sub-programs in the form of an MTS sub-program library. This term also has been loosely used as a synonym for "public file."

Library Programs GC

A collection of programs and/or sub-programs available to every user of the computing system. With MTS at WSU, library programs are usually available in public files and public library files. However, the term is usually used in reference to any program supplied and supported by the Computing and Data Processing Center, regardless of the form of availability.

Light Pen GC

A photo-electric device, about the size and shape of an ordinary pen, used as an adjunct to a CRT display unit. The user can pass the pen over the surface of the CRT to select images displayed on the screen. The light pen selection of an image can be used to initiate some programming action; for example, modification of the image being displayed.

Line - See Record. GC

Line File MTS

A direct access file consisting of ordered lines which may be up to 255 characters long and are each identified by a unique line number. See also Line Number.

Line Number MTS

A number which uniquely identifies each line in a line file. Externally, line numbers are in the range -99999.999 to +99999.999. Internally, MTS stores the line numbers as fullword integers. See also Line File.

Line Number Range MTS

A qualification put on a file, device, or pseudo-device name to indicate that only certain records from the file or device are of interest. The form of the qualification is "FDname(b,e,i)", where "b" is the number of the beginning record (line) of interest, "e" is the ending record, and "i" is the increment to determine the numbers of intermediate records between "b" and "e".

Line Printer - See Printer. GC

Linkage Editor 360

A program that takes object programs as produced by any of the language processors and transforms them into a form that loads very efficiently.

Listing GC

A printed copy of a program, a set of data, etc.

Literal GC

A symbol or a quantity in a source program that contains its value as an explicit part of its name.

Load GC

Usually, to read an object program into the computer's main storage, and prepare it for subsequent execution by carrying out the necessary relocation and resolution of external references. See also Load Map, Load Time, Loader, Object Program, Relocation, External Symbol Resolution, Unload.

Load Map GC

Listing of the memory locations at which programs and sub-programs have been loaded.

Load Module - See Object Programs. GC

Load Point GC

That point which is the beginning of the usable portion of a magnetic tape. It is the position of a tape after it has been rewound. See also Load Point Marker.

Load Point Marker GC

A strip of reflective material that marks the load point of a magnetic tape. It is automatically sensed by the tape drive hardware.

Load Time GC

(1) The CPU time required for an object program to be loaded into memory.
(2) The time at which the load process is taking place.

Loader GC

A program which reads object programs into main storage and prepares them for subsequent execution. See also Load.

Loading Routine - See Loader.

Local Time Limit MTS

A limit on the amount of CPU time that can be used by one segment of a complete computer job. In MTS, local time limits can be specified for single RUN commands. See also Global Time Limit.

Location - See Storage Location. GC

Logical Address - See Virtual Address. GC

Logical Carriage Control GC

Control of the vertical spacing of printed output as it is normally encountered at the programming level. Logical carriage control uses a set of carriage control characters whose action is related to a logical page which, in MTS, consists of 60 lines centered on each individual sheet of paper. Each logical carriage control character must be translated into its equivalent machine carriage control character before being sent to the the printer. The translation is done within the operating system and need not concern the programmer. See also Carriage Control Character, Machine Carriage Control.

Logical I/O Unit, Logical Unit, Logical Unit Number MTS

A name or number which may be used in a program to refer to an I/O device. The name is not connected with any particular file or device, so the program is independent of the actual file or device which is eventually used. The MTS logical I/O units are SCARDS, SPRINT, SPUNCH, SERCOM, GUSER, and 0 through 19.

Logical Record GC

A logical unit of data. See also Blocking.

Logical Unit - See Logical I/O Unit. MTS

Logical Unit Number - See Logical I/O Unit. MTS

Low-level GC

A term usually used when describing computer languages. Low-level languages are those close to machine language in syntax.

Low-order Position GC

The rightmost position in a number or word.

Machine Carriage Control GC

Control of the vertical spacing of printed ouput. Machine carriage control differs from logical carriage control in that it uses a set of

carriage control characters whose action is related to the actual hardware that causes the paper movement, rather than to a logical page. See also Carriage Control Character, Logical Carriage Control.

Machine Language GC

Refers to instructions written in the code which can be immediately executed by a computer without any translation.

Machine Word - See Fullword. GC

Macro, Macro Instruction GC

A source language instruction that is replaced, during translation, by a predetermined sequence of machine instructions, usually with provisions for parameter replacement.

Macro Library GC

A library which contains routines that are written in a macro language. With 360 computers, these are written in the 360-assembler macro language.

Magnetic Disk - See Disk. GC

Magnetic Drum - See Drum. GC

Magnetic Tape, Tape GC

A storage medium composed of a continuous strip of plastic material which is coated with a magnetic oxide. Data is recorded on this surface as a series of magnetized spots. For 360 use, the tape is recorded in either 7 or 9 parallel tracks along the length of the tape, and is wound around a plastic reel which normally holds 225, 600, 1200, or 2400 feet of tape. At WSU, all tapes must be 9 track; tapes assigned to users by the tape librarian are generally 2400 feet long. Tapes are useful for the storage of large volumes of fairly static data, and for shipping them by mail.

Magnetic Tape Density, BPI, or CPI, Tape Density GC

The number of bits per track or characters per inch of magnetic tape.

Magnetic Tape Dismounting, Dismounting, Tape Dismounting GC

Removing a reel of magnetic tape from a tape drive.

Magnetic Tape Drive, Drive, Tape Drive, Tape Unit GC

A device consisting basically of a mechanism capable of moving two tape reels at a high speed so that tape is wound from one reel onto the other. As the tape is transported between the two reels, it passes a read/write head which is used either to read from or write to the tape.

Magnetic Tape File Mark, End-of-file Mark, File Mark, Tape Mark GC

The end-of-file indicator on magnetic tape. For 360 use, it consists of a hardware-generated "tape mark gap" (a length of blank tape), and a special hardware-generated record which the hardware recognizes when reading the tape.

Magnetic Tape Label, Label, Tape Label GC

One or more blocks of information recorded at the beginning of a reel of magnetic tape. The first block contains identifying information that is used to check that the correct tape has been mounted by the computer operator. Subsequent blocks may contain dating or other information with reference to the actual identifying information in the first label block.

Magnetic Tape Mode, Mode, Tape Mode GC

Describes the physical properties of magnetic tape data; such as, density, parity, convertor on or off, and so on.

Magnetic Tape Mounting, Mounting, Tape Loading, Tape Mounting GC

The process of placing a reel of magnetic tape onto a tape drive, and preparing it for reading or writing.

Magnetic Tape Reel, Reel, Tape Reel GC

A plastic spool around which magnetic tape is wound.

Magnetic Tape Rewind, Rewind, Tape Rewind GC

To position a tape at the load point marker after reading from or writing to the tape. See Load Point Marker.

Magnetic Tape Ring - See File Protect Ring. GC

Magnetic Tape Unit - See Magnetic Tape Drive. GC

Main-line Program, Self-contained Program GC

A program that can be executed directly rather than having to be called from another routine.

Main Storage, Main Memory GC

The general purpose storage of a computer. Usually, main storage can be accessed directly by the operating registers in the central processor as contrasted to auxiliary storage which cannot. In general, all program-addressable storage from which instructions may be executed and from which data can be loaded directly into registers is called main storage. See also real memory, virtual memory.

Map - See Load Map, Memory Map. GC

Mask GC

A pattern of bits or characters that is used to control the selection, retention, or elimination of portions of another pattern of bits or characters.

Memory Map GC

A listing of all variables, constants, and statement labels in a program, and the storage location, usually relative to the start of the program, assigned to each. A memory map is typically produced by a compiler. See also Load Map.

Memory Protect GC

A feature that prevents the writing of data into specified areas of main storage, thus preventing a program from destroying the operating system or another user's program. In IBM 360 computers, the protected area is designated by a key in the program status word. Once the key is set, the program can only write into sections of storage that have the same key. See also Memory Protection, Protect Key.

Memory Protection, Storage Protection GC

An arrangement for preventing access to storage for either reading or writing or both. See also Memory Protect, Protect Key.

Michigan Terminal System, MTS MTS

An operating system designed to run on an IBM System 360 Model 67 computer. It is a multi-programming, multi-processor, interactive, time-sharing system that offers compatible batch and terminal facilities, and makes use of the special hardware features of the Model 67 in providing these facilities. MTS was developed by the University of Michigan Computing Center staff.

Mode - See Magnetic Tape Mode. GC

Modes of Operation MTS

(1) The MTS operating system allows computer access in several ways, for example, conversational terminal access, HASP batch access, remote batch access. Each of these can be referred to as a "mode of operation". See also Conversational Mode, HASP Batch Mode, Remote Batch Mode.
(2) The MTS command language has several modes of command operation. they are MTS command mode, edit mode, debug mode, or network mode. See also MTS Command Mode, Edit Mode, Debug Mode, Network Mode.

Modifier - See I/O Modifier. MTS

Module GC

A program unit that is discrete and identifiable with respect to compiling (the source module) and loading (the object module).

Monitor GC

Software or hardware that observes, supervises, controls, or verifies the operation of a system.

Mounting - See Magnetic Tape Mounting. GC

MSINK MTS

A pseudo-device used for output. MSINK stands for master sink, which is always the terminal in conversational mode or the printer in batch mode.

MSOURCE MTS

A pseudo-device used for input. MSOURCE stands for master source, which is always the terminal in conversational mode or the card reader in batch mode.

MTS - See Michigan Terminal System. MTS

MTS Command MTS

A command to the MTS operating system; an element of the MTS command language.

MTS Command Mode MTS

The state during which MTS is ready to accept either commands or data lines for the currently active file or device. At a terminal, MTS command mode is usually indicated by the presence of the "*" prefix character.

Multi-level Storage System GC

A system in which the most frequently used information is kept on higher speed storage devices, while less frequently used data is stored on slower auxiliary storage devices.

Multi-programming System GC

A system that supports more than one program in a state of activity at one time. MTS is one such system.

Multi-processing System GC

An operating system which runs on a computer that has more than one central processing unit. MTS is such a system. (The IBM 360 Model 67 at WSU is a dual-processor, that is, a duplex machine.)

Nesting GC

Pertaining to a set of programming instructions that contains a structure similar to itself; for example, a loop of instructions that contains another loop, and so on, possibly down through several levels.

Nine-track Magnetic Tape GC

Magnetic tape that has been written on a 9-track drive.

Nine-track Magnetic Tape Drive GC

A magnetic tape drive that writes 9 bits accross the tape, 8 data bits plus a parity bit.

Non-numeric Character GC

A character that is not a numeral.

Non-printing Graphics GC

Characters that can be represented within the computer, but for which there is no corresponding visible character on the printing device (such as, printer or Teletype). If one attempts to print such characters, a blank usually appears.

Object Code GC

Output from a programming language processor that is itself executable machine code or is suitable for processing to produce executable machine code.

Object Deck GC

A deck of cards which contain object code.

Object File GC

A file containing object code.

Object Language GC

The language in which the output from a programming language processor is expressed.

Object Module - See Object Program. GC

Object Program, Object Module GC

A program or sub-program in object code form.

Object Time - See Execution Time. GC

Octal GC

(1) Pertaining to the number system with a base of 8.
(2) Pertaining to a characteristic or property involving a selection, choice, or condition in which there are 8 possibilities.

Off-line GC

Descriptive of a system (and its peripheral equipment) in which the operation of the peripheral devices is not under the control of the central processing unit of the main computer.

OMR, Optical Mark Reader GC

A device capable of reading cards marked with pencil.

On-demand System GC

A system from which information or service is available immediately at the time of request.

On-line GC

(1) Describes peripheral devices under direct control of a CPU.
(2) Pertaining to a user's ability to interact with a computer.

Opening a File GC

The process of collecting the information necessary to begin I/O operations on a file.

Operating System GC

Software which controls the execution of computer programs and which may provide scheduling, debugging, I/O control, accounting, compilation, storage assignment, data management, and related services. The operating system at WSU is the Michigan Terminal System (MTS). See also Michigan Terminal System, OS, TSS.

Operator's Console, Console GC

The operator's console contains all the switches and indicators necessary for the operation of the central processor. It allows for such things as turning power on and off, starting and stopping the system, error status display, main storage and register display, etc.

Most consoles incorporate an alphanumeric display device (such as a typewriter or CRT) for communication between the operating system and the computer operator.

OS 360

The IBM operating system for IBM System/360 and 370 computers with more than 64K of main storage.

Output GC

The transferring of data from main storage to auxiliary devices. The term is also used to describe the devices and the data involved in such a transfer.

Output Device GC

Peripheral equipment to which output can be directed; for example, a printer, magnetic tape drive, disk file, etc.

Output Record GC

A unit of data intended for an output device.

Padding GC

The act of appending fill characters (usually blanks) to a record to bring it up to a specific length, for example, padding with trailing blanks.

Page GC

A unit of storage in a computer which is used by the addressing hardware to allow references to a location to be done as a displacement from a base address. For the IBM 360 Model 67, a page is 4096 bytes. See Paging.

Paging GC

Paging is a storage management technique which facilitates the moving of user programs in and out of high speed storage to slower peripheral storage. The various storage devices are divided into pages, and information is tranferred from main storage to auxiliary storage in page-sized blocks. Any subset of the pages assigned to a program can be in storage at one time. It is possible to allow a program to access more storage than is physically available in the system, for pages which are not currently required can reside on the auxiliary storage devices. With MTS, drums (and disks if the system is busy) are used for paging. Each user is provided with 6 million bytes of storage, even though the system has a total of only two million bytes of main storage. See also Virtual Memory.

Paper Tape GC

A computer I/O medium which is a strip of paper into which information can be recorded by means of a pattern of punched holes. Each character is recorded as a single row of holes across the width of the tape.

Paper Tape Punch GC

A device for punching holes in paper tape in response to signals transmitted from a computer.

Paper Tape Reader GC

A device that senses and translates the holes in paper tape into electrical signals for input to a computer.

Paper Tape Transmission Code 360

A character transmission code for use between a terminal device and a computer. Each character is uniquely represented by a 6-bit number with a seventh bit generated for parity. The parity bit is used as a check bit.

Parameter GC

A unit of variable information passed by a user to the operating system or from one program to another.

Parity Bit GC

A check bit used to assure the validity of stored data (whether the data is in main storage or tape or wherever). In general, the parity bit is an extra bit attached to a small part of storage (with IBM 360 computers, usually the byte). When data is stored, the parity bit is set so that the sum of all the bits set is either even or odd (usually odd). When the data is read back, the associated parity bit is checked to be sure that the odd or even relationship still holds - if not, the hardware signals an error condition.

Password MTS

A user controlled character string which must be given (together with an ID code) with the SIGNON command in order to gain access to the computing system. The password is part of the security mechanism preventing people from using ID's, and therefore accounts, that aren't their own. See also Password Card.

Password Card MTS

In batch mode, the password should always be given on the card follwing the SIGNON command. This password card is not printed with the rest of the job's output, and thus the security of the password is maintained. The password card should be punched with keypunch printing disabled.

PDN - See Pseudo-device Name. GC

Peripheral Unit GC

A machine which can be operated under computer control. Peripheral equipment includes input devices (such as, card readers, or magnetic tapes), output devices (line printers, plotters), and storage devices (disks, drums).

Permanent File MTS

Any file that is created by a user and which does not have the scratch file character (usually a minus sign "-") as the first character of its name. Such files are automatically saved until they are explicitly destroyed with a DESTROY command. See also File, Line File, Sequential File, Public File, Library File.

Permit Code MTS

A code attached to a file which controls access to the file.

Physical Record, Block GC

A unit of information as it appears on some storage medium. See also Logical Record, Record.

PIL, PIL2 GC

Pittsburgh Interpretive Language; an interactive, procedure-oriented language designed for terminal use.

PL/I GC

Programming Language I; a high-level programming language suitable for both scientific and business use.

PL/I Library MTS

A file *PL1LIB containing the object decks of subroutines frequently referenced by PL/I programs.

Plotter GC

An electro-mechanical device used to produce computer generated drawings, graphs, etc.

Positional Parameter GC

A parameter whose identity is determined by its particular position in a sequence of parameters; for example, in MTS, the ccid code must be the first parameter given with the $SIGNON command.

Precision - See Single Precision, Double Precision. GC

Prefix Character MTS

The first character of each line on conversational terminals is called the prefix character. The most important prefix characters are printed by MTS and are: "*" which is issued when the terminal is in MTS command mode, ">" which is issued with output from the LIST and COPY commands, and "?" which is issued to prompt for responses to queries. Output and input lines for user programs are also preceded by prefix characters, normally the blank.

Preprocessor GC

A term used to refer to a program which transforms data in some way before it is used by another program.

Printer GC

An output device which converts data into printed form on paper.

Printer Plot GC

Graphical output produced on a printer.

Priority GC

A parameter determining, in part, the order in which batch jobs are processed. See also HASP Priority.

Private File - See Permanent File. MTS

Problem State GC

When a computer is executing the instructions of a problem program (as opposed to the operating system), it operates in what is called the problem state. In this state, all I/O instructions and a subset of the control instructions are invalid. This ensures that one user's program cannot execute an instruction that affects other users' programs. See also Supervisor State.

Procedure-oriented Language GC

A higher-level language suitable for expressing algorithms quite directly.

Processor - See Central Processing Unit. GC

Program GC

(1) A series of instructions in some computer language.
(2) To design, write, and test such a series of instructions.

Program Interrupt 360

A specific category of interrupts which is caused by the execution or attempted execution of an illegal instruction or an illegal use of data for an instruction.

Program Segment GC

Usually a set of programming instructions which have some sort of logical or physical connection, but which do not form a complete program as defined by the particular programming language used.

Program Status Word, PSW 360

A doubleword containing information necessary to describe the status of the computer.

Project MTS

A designation of a group of MTS ccids with a common account to be billed.

Protect Key GC

A flag in the program status word on the System 360 which designates the parts of main storage into which a program may write. A program can write into only those areas which have the same key. Thus, if different users are given different protect keys, then one user can in no way affect another user. See also Memory Protect, Memory Protection.

Pseudo-device MTS

Logical device classes which are used in place of actual devices so that programmers need not concern themselves with the physical devices their programs use. The correspondence between the logical device class and the physical device used is made by the operating system. See also Pseudo-device Name.

Pseudo-device Name MTS

The name of a pseudo-device. For example, in MTS batch mode, the pseudo-device name referring to the printer is *MSINK*. In MTS, each pseudo-device name starts and ends with an asterisk (*). Some names are predefined by the system, and others can be defined by the user.

PSW - See Program Status Word. 360

PTTC - See Paper Tape Transmission Code. 360

Public File MTS

A file which has a name starting with an asterisk (*), CCAP:, HELP:, or DEMO:, which can be read by all users of the computing system. Computing center staff use public files to maintain programs and subprograms of general interest.

Public Library File MTS

A public file containing object modules organized as an MTS sub-program library. The term has been used (loosely) as synonymous with "public file". In this context, "library" refers to those programs and sub-programs provided by the Computing Center for general use, rather than to the actual format of the file.

PUNCH MTS

A pseudo-device name referring to the system card punch.

Punch - See Keypunch, Paper Tape Punch, Reproducing Punch. GC

Punched Card - See Card. GC

Punched Output GC

Computer output produced by some sort of punch device, such as, a card punch or a paper tape punch.

Queue - See Batch Queue. GC

Rack Number MTS

A number assigned to the physical storage location of each magnetic tape in the Computing Center's storage racks.

Random Access - See Direct Access. GC

Raw Data GC

Data which has not been transformed or otherwise processed.

RC - See Return Code. GC

Read GC

To acquire or interpret data from a storage device or any other source.

Reader GC

A device which converts information in one form of storage to information in another form of storage. See also Card Reader, Paper Tape Reader.

Read-only Storage GC

Describes storage from which data can be read but not changed.

Read-write Head GC

An electromagnetic device used to read data from or write data onto a magnetic medium such as magnetic tape, disk, or drum.

Real Address GC

An address that indicates, in the machine code address numbering system, the exact storage location where the referenced operand can be found or stored.

Real Memory GC

A term referring strictly to the primary storage medium (usually main storage) of a computer. See also Virtual Memory.

Real Time GC

(1) Pertaining to the actual time taken for a physical process.
(2) Pertaining to the performance of a computation during the actual time that a related physical process is taking place, often in order to monitor and guide the process.

Real Variable GC

A variable whose value is numeric and stored in floating point form.

Receipt Card - See S-8 Card. MTS

Record GC

A collection of related items of data, treated as a unit. See also Logical Record, Physical Record.

Record Density - See Density.

Record Format GC

The organization of a record.

Record Gap, Inter-record Gap, Interblock Gap GC

A length of blank tape (0.6 inches long for 9-track tape, 0.75 inches for 7-track) used to separate physical records on magnetic tape.

Record Index GC

A quantity that points to the next record to be processed.

Record Length GC

The number of units (bytes with IBM 360 computer) of data in a record.

Reel GC

The plastic spool about which magnetic tape is wound.

Re-entrant GC

Descriptive of a program that doesn't modify itself in any way. In a time-sharing computer system, a single copy of a re-entrant program can be used by many users "at the same time". Since such a program doesn't modify itself, one person's use of the program cannot affect another person's use.

Register GC

Part of the central processing unit of the computer which holds information to be subjected to arithmetic and logical operations. In IBM 360 computers there are two types of registers; general registers for logical and integer numeric operations, and floating-point registers for floating-point numeric operations.

Relative Addressing GC

A system of programming in which instructions are written so that they refer to some base location (usually the start of the program) rather than to an absolute storage address. When the program is executed, the base address is added to the address component of each instruction in order to create absolute addresses.

Remote Batch Mode GC

Refers to the submission of non-interactive jobs to the computer from card readers not located with the computer. See also HASP, HASP Batch Mode, HASP Priorities.

Remote Terminal GC

An I/O device not located with the computer. The term usually refers to conversational terminals, but is used to describe batch (non-interactive card reader/printer) terminals also.

Reproducing Punch GC

A machine which reads a deck of cards and automatically produces a duplicate copy.

Resident System GC

A term used to denote that part of the operating system which resides in main storage at all times.

Resolve External Symbol Resolution. GC

Resolve Library References - See External Symbol Resolutuion. GC

Response Time GC

The real time taken to complete a requested computation. The term is usually used in reference to calculations done at conversational terminals.

Restart GC

To re-establish the execution of a program.

Return Code, RC GC

A number returned by sub-programs that indicates the success or failure of some part of the task done by the sub-program. In IBM 360 computers, it is usually returned in general register 15 and is a multiple of 4 in value. The return code is used by FORTRAN to set up multiple returns from subroutines (that is, RETURN 1, RETURN 2, etc.).

Re-usable GC

The property or attribute of a program which makes it self initializing so that it can be used more than once, or by more than one user, but not necessarily at the same time. See also Re-entrant, Serially Re-usable.

Rewind - See Magnetic Tape Rewind. GC

Right Justify - See Justify. GC

Ring - See File Protect Ring. GC

Round-off Error GC

An error resulting from the procedure of deleting some of the less significant digits of numbers and, possibly, applying some rule of correction.

Routine GC

A program or sub-program. See also Program, Sub-program.

Run Time - See Execution Time. GC

Save Area 360

The storage area belonging to a calling program and used by the called program to save and later restore general registers and other information.

SCARDS MTS

A logical I/O unit which defaults to *SOURCE*. SCARDS stands for system cards.

Scratch File - See Temporary File. MTS

SDS - See Symbolic Debugging System. MTS

Secondary Storage - See auxiliary storage. GC

Self-contained Program - See Main-line Program. GC

Sense Bytes 360

Coded information, generated by hardware, concerning the status of a device. For example, sense bytes give information about the cause of tape reading errors.

Separator - See Delimiter. GC

Sequential File GC

(1) A data file in which the records must be accessed in a fixed order, one after the other. Magnetic tape files are sequential files.
(2) A particular type of MTS file organized as in (1).

SERCOM MTS

A logical I/O unit which defaults to *MSINK*. System and program error messages are normally directed to SERCOM. SERCOM stands for system error comments.

Serially Re-usable GC

Describes a program that can be used again and again from the beginning without having to be reloaded into main storage, but which only one user can use at a time. See also Re-entrant, Re-usable.

Seven-track Magnetic Tape GC

Tape that has been written by a 7-track magnetic tape drive. See also Seven-track Magnetic Tape Drive.

Seven-track Magnetic Tape Drive GC

A magnetic tape drive that records data in 7 channels running lengthwise along the tape. Six bits across the tape are used to hold data and one bit is for parity.

Severity Code GC

An indicator associated with an error message produced by a language processor and which indicates the seriousness of the error.

Signon id - See Ccid. MTS

Single Precision GC

The use of a single computer word to hold the results of calculations. See also Double Precision.

SINK MTS

A pseudo-device name referring to an output file or device. It defaults to the terminal in conversational mode and the printer in batch mode. However, its designation may be changed with the SINK command.

Snark MTS

A condition of multiple program interrupts in MTS which requires the system operator to take special action to sign off the affected user.

SNOBOL GC

A symbol manipulation language which is useful for language translation, program compilation, and combinatorial problems.

Software GC

Programs, as opposed to hardware. See also Hardware.

Sorter GC

A machine used for sorting punched cards into some desired sequence according to information punched on the cards.

SOURCE MTS

A pseudo-device name referring to the input file or device from which MTS commands and data for the currently active file are taken. It defaults to the terminal in conversational mode and to the card reader in batch mode. However, the *SOURCE* file or device can be changed with the SOURCE command.

Source Deck GC

A card deck on which a program has been punched in a computer language other than machine code.

Source Language, Symbolic Language GC

Any programming language which cannot be directly processed by a computer, but requires translation into an object program consisting of machine code instructions, for example, PL/I, FORTRAN, ALGOL.

Source Program GC

A program as written in a source language.

Special Character GC

A character that is neither alphabetic, numeric, nor a blank.

SPOOLING GC

(Simultaneous Peripheral Operations On-line) A process whereby the system reads all cards for a batch job before starting the job, and saves the printed output until the job is finished before starting to print it. This is in distinction to having the card reader and printer attached to the job for the duration, thereby forcing all jobs to proceed at the rate of the slowest component.

SPRINT MTS

A logical I/O unit which defaults to the pseudo-device *SOURCE*. SPRINT stands for system print unit.

SPUNCH MTS

A logical I/O unit which defaults to the pseudo-device *PUNCH*. It is the system punch unit.

Stacker - See Card Stacker. GC

Station GC

The location of a computer input/output device, as in, for example, a remote batch station.

Storage GC

Any medium for storing information so that it is available to a computing system when required. See also main storage, auxiliary storage, Virtual Memory.

Storage Location GC

A particular position within the main storage of a computer.

Storage Protection - See Memory Protection. GC

Store GC

To enter or retain data in a storage device.

String GC

A sequence of information elements of a similar type (such as bits, bytes, characters), treated as a single item of data.

Sub-program, Subroutine GC

A part of a larger program written in such a way that allows it to be called from several places in the main program. A sub-program may be translated into machine language and tested independently of the main program.

Subroutine - See Sub-program. GC

Supervisor, Supervisor Program GC

A program (or system of programs) that is always available in the computer's main storage for controlling the sequencing, setup, and execution of all jobs entering the computer. A supervisor is usually one part of an operating system. UMMPS is the supervisor for MTS.

Supervisor Call 360

A special instruction used by a program to force an interruption of the job in progress through a transfer of control to the supervisor. This action is needed for input/output operations and other operations that place special demands on the resources of the computer system. A supervisor call causes a transfer from the problem state to the supervisor state. See also Problem State, Supervisor, Supervisor Interrupt, Supervisor State.

Supervisor Interrupt 360

A transfer of control to the supervisor because of some exceptional condition. This may arise in the program being executed (for example, through division by zero), or in some other process that is being handled concurrently (for example, input/output). See also Problem State, Supervisor, Supervisor State, Supervisor Call.

Supervisor Program - See Supervisor. GC

Supervisor State GC

A special condition of the central processor that permits execution of all instructions in the repertoire of the machine, including control and input/output instructions not directly available to the user. See also Problem State.

Supervisor Task Number, Task Number MTS

The number which the supervisor (operating system) assigns to each task currently running in the system. This number provides a means of uniquely identifying each task.

Symbolic Debugging System MTS

A program (entered via debug mode) which aids in the conversational debugging of programs. See also Debug Mode.

Symbolic Language - See Source Language.

Symbol Table GC

A list of the symbols used in a program, together with information about their attributes and their use.

System GC

The computer hardware and/or software necessary for the execution of information processing jobs. See also Operating System, Supervisor.

System Interrupt - See Supervisor Call.

System Library GC

A collection of programs and sub-programs available to all users of a particular computer system. In MTS, it is the file *LIBRARY which is normally searched last during the program loading process.

S-8 Card MTS

A special card which must precede all other cards in a batch job. It signals the beginning of a new job. This card is so named because column 1 contains an S (0-2 punches) and an 8 (8 punch). This card is also called a Receipt card since it must be presented (as a receipt) to pick up output from the job.

Tape - See Magnetic Tape, Paper Tape. GC

Task Number - See Supervisor Task Number. MTS

Temporary File, Scratch File MTS

A disk file created for the duration of, at most, a single computer job. The names of temporary files begin with the scratch file character (usually a minus sign "-"). Temporary files need not be explicitly created with the CREATE command unless they are to be large or sequential files; MTS creates a temporary file when it is first referenced, if it does not already exist. Temporary files are automatically destroyed at sign-off time.

Terminal GC

A device attached to a computer system for the input or output of programs and data. See also Conversational Terminal.

Terminal Job GC

A computer job submitted from a conversational terminal.

Terminal Mode - See Conversational Mode, Remote Batch Mode. GC

Time-Sharing GC

A method of operating a computer system so that the time of the central processor is shared among a number of users by giving short bursts of activity to each in turn. In suitable circumstances, each user can operate as if the whole system were dedicated to him.

Time Slice GC

In a time-sharing system such as MTS, the period during which a program is using the CPU. See also Wait.

Track GC

The portion of a moving storage medium, such as a drum, tape, or disk that is accessible to a read head.

Translator GC

A program which converts statements written in one programming language to the format of another programming language.

Trap - See Interrupt. GC

Truncate GC

To suppress those digits of a number which are not significant according to some predetermined requirement. No correction is applied to the remaining digits. Truncation is also used in MTS to eliminate any unnecessary space in a file.

Truncation Error GC

(1) An error arising from inaccuracy in truncating a numeric result.
(2) In some contexts, an error arising from using an approximate formulation of the original problem.

TSS 360

Time Sharing System; the IBM operating system designed to offer time-sharing services on the IBM 360/67 computer.

UMIST MTS

University of Michigan Interpretive String Translator; an interactive text processing language.

Unit - See Device. GC

Unload GC

(1) See Magnetic Tape Dismounting.
(2) To remove a program from main storage.

Update GC

To modify a master data file with current information according to some specified procedure. The term is also used to refer to the current information itself, as used in the modification of the file.

User GC

A general designation for anyone who uses the facilities of a computer.

User File - See Permanent File. MTS

Utility Program GC

A standard program used for a frequently occurring task; for example, sorting, conversion between binary and decimal representation, etc.

Variable GC

A quantity that can assume any of a given set of values.

Vertical Redundancy Check GC

A parity check on the bits across the width of a byte or word.

Virtual Address, Logical Address GC

An address generated by a program which references virtual memory and must therefore be translated into a real storage address when it is used.

Virtual Memory GC

Memory which appears to the user as though it were all main storage, although it may actually be at any time partly in main storage, partly on a drum, partly on a disk, and even partly not allocated anywhere (if defined, but not referenced), depending on the competing demands for storage by other users. See also main storage, Paging.

Volume GC

A unit of peripheral storage, such as a disk pack or magnetic tape.

Wait GC

The condition of a job in a computer system when it has been started, but is not currently being worked on by the central processor. A common reason for this condition is that the job is waiting for the completion of an action by some other component of the system, for example, a disk storage unit or other peripheral device. In a multiprogramming system, the central processor interleaves work on a number of jobs with the waiting periods of any particular job.

WATFOR GC

A FORTRAN IV compiler written at the University of Waterloo. It is characterized by rapid compilation, extensive error checking at both compile and run times, and slow execution, when compared with other FORTRAN compilers. WATFOR does not generate an object deck. See also FORTRAN G Compiler, FORTRAN H Compiler.

WATFOR Library MTS

A set of sub-programs, in the public file *WATLIB, automatically available to users of the WATFOR compiler.

Word - See Fullword. GC

9-edge GC

The lower edge of a punched card. This is the edge adjacent to the row of 9-punches on the card.

12-edge GC

The upper edge of a punched card. This is the edge adjacent to the row of 12-punches on the card.

7-track Magnetic Tape - See seven-track Magnetic Tape. GC

9-track Magnetic Tape - See nine-track Magnetic Tape. GC

INDEX

April 1974

WSU CDPC

Reader's Comment and Update Request Form

MTS--THE SYSTEM
Vol. 1, third edition, revised

If you wish to receive any susequent updates to this volume, please fill in this form, fold as shown on the reverse side, seal or staple and mail. Your comments, if you desire to make any will be appreciated.

Suggestions and Comments

Date_____

Name_____

Department _____

Address_____

_____

_____

April 1974

fold here

Documentation Librarian
Computing and Data Processing Center
Wayne State University
5950 Cass Avenue
Detroit, Michigan 48202
USA

fold here