DOCUMENT RESUME

ED 060 629 EM 009 635

AUTHOR	Mosmann, Charles; Bork, Alfred M.
TITLE	Teaching Conversations with the XDS Sigma 7. System Users Manual.
INSTITUTION	California Univ., Irvine. Physics Computer Development Project.
SPONS AGENCY	National Science Foundation, Washington, D.C.
PUB DATE	26 May 71
NOTE	71p.
EDRS PRICE	MF-$0.65 HC-$3.29
DESCRIPTORS	*Computer Assisted Instruction; Computer Programs; *Manuals; Physics Instruction; *Programed Instruction; *Programing; Programing Languages
IDENTIFIERS	Metasymbol; Sigma 7

ABSTRACT

This manual is intended as a reference handbook for use in writing instructional dialogs on the Sigma-7 computer. The concern is to give concise information which one would need to write and debug dialogs on this system. Metasymbol, the macro-assembly program for the Sigma-7, is described. Definitions of terminology, legal forms descriptions of current commands, and examples are given. Basic, introductory information on getting dialogs into the computer, assembling and debugging them, and in preparing them for student use, makes up most of this manual. (RB)

ED 060629

pcdp

TEACHING CONVERSATIONS WITH THE XDS SIGMA 7

System Users Manual

Charles Mosmann
Alfred Bork

Physics Computer Development Project
University of California, Irvine
Irvine, California 92664

May 26, 1971

EM 009 635

CONTENTS

INTRODUCTION
1. WRITING DIALOGS
Program Fo
Line Forma
Character
Labels and
2. DESCRIPTION OF C
Displaying
Accepting
Analyzing
Manipulati
Manipulati
Manipulati
Constant a
Other Comm
3. GETTING IT ON TH
Using the
Using Edit
Using Meta
Using Delt
Generating
Sectioning
Overlaying
Student Us

dp

XDS SIGMA 7

ject

t, university of california, irvine, 92664

CONTENTS

INTRODUCTION

1. WRITING DIALOGS
   - Program Format
   - Line Format
   - Character String Constants
   - Labels and Location Symbols
2. DESCRIPTION OF COMMANDS
   - Displaying Information
   - Accepting Information
   - Analyzing Input
   - Manipulating Strings
   - Manipulating Numbers
   - Manipulating Counters
   - Constant and Parameter Storage
   - Other Commands
3. GETTING IT ON THE MACHINE
   - Using the Terminal
   - Using Edit
   - Using Metasymbol
   - Using Delta
   - Generating a Load Module
   - Sectioning a Program
   - Overlaying
   - Student Use

APPENDIX 1 - Examples
APPENDIX 2 - References and Acknowledgements
APPENDIX 3 - A Final Word to the Reader

INTRODUCTION

This manual is
person writing
document, calle
System Descript
presents a bett
our concern is
person will nee

The first chapt
tions of termin
commands curren
contains introd
getting dialogs
preparing them
tutorial in int
more complete d

The system desc
expands. If yo
be sure to fill
up-to-date on i

INTRODUCTION

knowledgements

e Reader

This manual is intended to be used as a reference handbook by the person writing instructional dialogs using the Sigma-7. Another document, called "Teaching Conversations with the XDS Sigma-7: System Description," presents an overview of the system as a whole and presents a better introduction to the subject than this manual. Here our concern is to make a concise presentation of the information a person will need while learning to write and debug dialogs.

The first chapter contains the rules on legal forms and some definitions of terminology. Chapter 2 contains descriptions of each of the commands currently available, with examples of their use. Chapter 3 contains introductory material on the programs and routines used in getting dialogs into the computer, assembling and debugging them, and preparing them for student use. This information is introductory and tutorial in intent and does not by any means replace the need for the more complete descriptions in the appropriate XDS manuals.

The system described here continues to change and grow as its use expands. If you are a user of this system and not just an observer, be sure to fill out the form on the last page so that you can be kept up-to-date on improvements as they occur.

CHAPTER 1

WRITING DIALOGS

An instructional dialog using the commands defined here is actually a program written for Metasymbol, the macro-assembly program for the Sigma-7. The dialog must therefore adhere to Metasymbol conventions for formats and the special use of symbols. In addition, some conventions have been established within the dialog system itself. These topics are the subject of this chapter.

1. Program Format

The first statement of the program must be

```
        SYSTEM     DIALOG
```

The SYSTEM command (preceded by at least one space) directs the assembler to select a file containing the commands that are to be valid during this assembly; DIALOG is the name of the file containing the commands described here.

After the SYSTEM statement but before the first executable command, the program must have one or the other of these statements:

```
        NAME     arg
        START
```

Either of these commands will cause the system to introduce some initializing procedures into your program, which are to be executed

before your first comm
argument you supply, c
quotes) to your progra
It is particularly val
for later reference; t
records so that there
those of other instruc

The last statement of

END   DIAL

This indicates to Meta
process. The argument
routine is to be used
for an ID if the resta
Other arguments are po
ment in the program, i
that will be executed
of the introductory ar
only with caution and

Additional information
in the sections on seg
to the author of very

2. Line Format

Each line may contain

mands defined here is actually
macro-assembly program for the
dhere to Metasymbol conventions
mbols. In addition, some con-
the dialog system itself.
chapter.

ust be

east one space) directs the
the commands that are to be valid
ame of the file containing the

the first executable command,
er of these statements:

the system to introduce some
gram, which are to be executed

before your first command. In addition, NAME assigns a name (the argument you supply, of not more than four characters enclosed in quotes) to your program. This simply individualizes the program. It is particularly valuable when student responses are saved on disk for later reference; the name of the program individualizes these records so that there is no possibility of getting them confused with those of other instructional programs.

The last statement of the program must be

```
END     DIALOGUE
```

This indicates to Metasymbol that this is the last statement it is to process. The argument "DIALOGUE" indicates that a standard beginning routine is to be used when the program is run. (It asks the student for an ID if the restart facility is used somewhere in the program.) Other arguments are possible: if the argument is the label of a statement in the program, it is understood that this is the first statement that will be executed when the program runs. This short-circuits all of the introductory and initializing instructions and should be used only with caution and a full understanding of the implications.

Additional information about program format will be found in Chapter 3 in the sections on segmenting and overlaying; these will be of interest to the author of very large programs.

__2. Line Format__

Each line may contain four __fields__, separated by one or more blanks:

LABEL COMMAND ARGUMENT COMMENT

Because blanks are the delimiters of the fields, it is clear that blanks are never allowed <u>within</u> the fields (except within character strings, as described below). The label field may be omitted; if it is present, it must begin in the <u>first</u> position on the line. If that position is blank, it is assumed that there is no label and the command field follows immediately. Labels are attached to statements so that they can be referred to elsewhere in the program; label formats are described below. The next field is the command; either one of the commands listed here or one of those defined in the Symbol/Metasymbol manual. The command field must be present; it is the command which indicates what it is that you want the computer to do. The third field is the argument. The parameters necessary to complete the meaning of the statement are entered here, usually separated by commas--again, without blanks, since a blank indicates the end of the field. The fourth field is for comments and is ignored by the assembly program. (If the command requires no argument, then everything past the command field is ignored. The line (if it is entered from a terminal) can be ended at any point by a carriage return.

There is a single exception to all of this. If the first character on the line is an asterisk, the entire line is ignored. Such lines are considered "comments" and can be of value in making your program clearer to read and understand.

There are thus no restr
in which column or posi
program easier to read
label in columns 1-8, c
20-39, comments beginni
tab stops) make it easi
the program from a term

<u>3. Character string co</u>
A large part of any ins
of characters which are
pare with student respo
quotes. They may conta
to the "no blank" rule.
a string, you must use

'#8()+-&='

' '

'YOU''RE RIGH

<u>4. Labels and Location</u>
The programmer may assi
variable storage locati
common way of attaching
label in the first fiel
attached to parameter s
COUNTER, STRING, DEFNUM
using the label field.

COMMENT

fields, it is clear that
is (except within character
field may be omitted; if it
osition on the line. If
at there is no label and the
ls are attached to statements
e in the program; label
ield is the command; either
f those defined in the
ield must be present; it is
hat you want the computer to
he parameters necessary to
e entered here, usually
ks, since a blank indicates
is for comments and is
e command requires no argument,
s ignored. The line (if it
at any point by a carriage

is. If the first character
ine is ignored. Such lines
value in making your program

There are thus no restrictions about what information can appear in which column or position on the line. Nevertheless, it makes a program easier to read if some such conventions are adhered to; perhaps, label in columns 1-8, command in columns 10-19, argument in columns 20-39, comments beginning in 40. Tab stops (see Chapter 3 for using tab stops) make it easier to adhere to these conventions when entering the program from a terminal.

3. Character string constants

A large part of any instructional dialog program is made up of strings of characters which are to be typed for the student to read or compare with student responses. Such strings must be enclosed in single quotes. They may contain any characters; they are thus an exception to the "no blank" rule. If you want to place a single quote inside such a string, you must use two single quotes. A few examples may help:

```
'#8()+-&='
' '
'YOU''RE RIGHT!'
```

4. Labels and Location Symbols

The programmer may assign names to instructions, constants, and variable storage locations so that he can refer to them. The most common way of attaching such symbols to locations is by the use of a label in the first field of a statement. Symbolic names must also be attached to parameter storage. There are special commands (DEFINE, COUNTER, STRING, DEFNUM, etc.) described below which do this without using the label field. The label or location symbol may consist of

any combination of letters and digits except all digits. (The advanced student should be cautioned against using any labels beginning with character "#" since this is used internally in labels in the macro definitions, and so might lead to conflicts.)

One location symbol used in the program may be of interest to the user. The input buffer, containing the most recent message typed by the student is called #IN and may be referred to by that name. It is defined by the system and must not be defined by the user.

CHAPTER 2

DESCRIPTION OF COMMANDS

This chapter contains de
available. They are arr
the following page lists

In general, the descript

NAME (SYNONYM1

Description of
When several a
described.

Example 1

desc

Example 2

desc

Examples are given of ea
forms, except where the
If terms used in the des
them defined above in Ch

1. Displaying Informati

Commands for displaying

WRITE (PRINT,
OUT
SKIP (SAUTEZ)
GRAPH
PLOT
NUMOUT (OUTNUM
NUMWRITE (WRIT
OUTABLE

except all digits. (The advanced
ing any labels beginning with
nally in labels in the macro
licts.)

m may be of interest to the
e most recent message typed
e referred to by that name.
ot be defined by the user.

CHAPTER 2

DESCRIPTION OF COMMANDS

This chapter contains descriptions of all the commands currently available. They are arranged by functions; however, the index on the following page lists them in alphabetical order.

In general, the descriptions will have the following format:

> <u>NAME</u> (SYNONYM1, SYNONYM2...)
>
> Description of the logical function of the command. When several alternative forms exist, each will be described.
>
> Example 1:
>
> description of function of example 1
>
> Example 2:
>
> description of function of example 2

Examples are given of each form of a command, when there are multiple forms, except where the meaning appears too obvious to warrant it. If terms used in the description are unfamiliar to you, you may find them defined above in Chapter 1.

1. Displaying Information

Commands for displaying information to the student.

WRITE (PRINT, ECRIVEZ, SHOW, TYPE)
OUT
SKIP (SAUTEZ)
GRAPH
PLOT
NUMOUT (OUTNUM)
NUMWRITE (WRITNUM)
OUTABLE

2. Accepting Information

Commands for accepting information from the student.

INPUT (ACCEPT, ACCEPTEZ)
INBELL

3. Analyzing Input

Commands for analyzing student input

IF (SI)
IFONLY
IFNOT
IFYES
IFNULL
IFBEFORE
IFAFTER (IFNEXT)
IFNEXTO (IFAFTERO)
IFTERMS
ALLWRONG
NOTERMS
IFKE
IFPE
IFFILTH
TO (OTHER, AUTRE, B, JUMP)

4. Manipulating Strings

Commands to manipulate strings

NOBLANK
DELETE (REMOVE)
DELETEALL
REPLACE (SUB, SUB:FOR)
SUBALL
ADDAST
MOVE (MOVEZ)
PTRMOVE
APPEND

5. Manipulating Numbers

Commands to manipulate numbers

NUMBER
SCAN
SCAN#
IFNUMEX
IFNOTNUM
AROUND
BETWEEN
RANDOM
RANDOMR

6. Manipulating Counters

Commands to manipulate c

BUMP (INCREASE
DECREASE (SUBl
RESET (ZERO)
CTARITH
ADDCOUNT
CTWRITE
CTOUT
SWITCH

7. Constant and Paramete

Directives for constant

DEFINE (DEFINE
COUNTER (COMPT
SET
STRING
TEXT
DEFNUM
STORENUM
DEFCOMP
STORECOMP
DEFTABLE
STACK

8. Other Commands

Other commands and direc

SYSTEM
NAME
START
END
STOP (HALT)
FINALE (EPILOG
ENTRY
SAVE (KEEP, IN
SAVEID
FORTRAN

student.

IFNOTNUM
AROUND
BETWEEN
RANDOM
RANDOMR

6. Manipulating Counters

Commands to manipulate counters.

BUMP (INCREASE, ADD1, AUGMENT)
DECREASE (SUB1)
RESET (ZERO)
CTARITH
ADDCOUNT
CTWRITE
CTOUT
SWITCH

7. Constant and Parameter Storage

Directives for constant and parameter storage.

DEFINE (DEFINEZ)
COUNTER (COMPTEUR, C)
SET
STRING
TEXT
DEFNUM
STORENUM
DEFCOMP
STORECOMP
DEFTABLE
STACK

8. Other Commands

Other commands and directives.

SYSTEM
NAME
START
END
STOP (HALT)
FINALE (EPILOGUE, EPILOG)
ENTRY
SAVE (KEEP, INFO)
SAVEID
FORTRAN

Ex. 2 OUT MES8

...

MES8 STRING 'IS THE SQUAREROOT OF'

Causes "is the squareroot of" to be typed.

Ex.3 OUT MES8,T3

Causes a branch to T3 after the string at MES 8 is printed.

Remember to supply connecting spaces on either the WRITE or the following OUT.

SKIP (SAUTEZ). Generates one or more blank lines at the terminal. The argument indicates the number of lines. If no argument is given, one line is skipped..

Ex. 1 SKIP 5

Causes five blank lines to appear.

Ex. 2 SKIP

Causes one blank line.

GRAPH. Displays data in the form of a point graph. It requires three arguments; the location of the horizontal displacements (X); the location of the vertical displacements (Y); the number of points to be plotted or the location of that number. All numbers will be scaled. Storage of the values is the user's responsibility; STACK may be useful in this program. If the values are generated in a FORTRAN subroutine (say as an array), space must be reserved in the dialog program.

Ex. 1 GRAPH X,Y,20

Graph 20 points, taking the coordinates from X and Y.

Ex. 2 GRAPH TA1,TA2,CNTR

Graphs the number of points specified in CNTR, taking the coordinates from TA1 and TA2.

At the beginning of a graph is a header giving the maximum and minimum values of the two arrays. The user can control scaling on a series of graphs by placing suitable values himself in the two arrays.

EROOT OF'

root of" to be typed.

after the string at MES 8 is

nnecting spaces on either the
ng OUT.

ates one or more blank lines at
gument indicates the number of
nt is given, one line is skipped..

es to appear.

e.

a in the form of a point graph.
guments; the location of the
ents (X); the location of the
ts (Y); the number of points
location of that number. All
ed. Storage of the values is
ility; STACK may be useful in
e values are generated in a
say as an array), space must
ialog program.

ing the coordinates from X and

TR

points specified in CNTR,
es from TA1 and TA2.

a graph is a header giving the
values of the two arrays. The
ling on a series of graphs by
ues himself in the two arrays.

PLOT. Generates a point plot as specified in the arguments. The first argument is the location of the values to be plotted; the second argument is the number of values or the location of that number. All of the values will be scaled and each value will represent the amount of horizontal displacement for each point. The difference between a plot and a graph is that the plot increments the vertical component uniformly. A header on a plot gives maximum and minimum displacement.

Ex. 1 PLOT EX,20

Will generate a plot of 20 values stored at EX.

NUMOUT (OUTNUM). Converts to character type, and prints, floating point numbers. It prints one number (specified in the argument) on the same line as the last written record (i.e., no carriage return).

Ex. 1 NUMOUT DISTANCE

Will print the number stored in DISTANCE, on the current line.

NUMWRITE (WRITENUM). Converts to character type and prints floating point numbers. It starts a new line and will print from one to four numbers, as specified in the argument(s).

Ex. 1 *NUMWRITE TIME,ENERGY,MASS

Starts a new line and prints the three numbers stored in TIME, ENERGY, and MASS.

OUTABLE. Prints the contents of a table. (The table could have been defined using DEFTABLE and filled using STACK.) The first argument is the name of the table; the second argument is the number of values to be printed (or ALL); the third (optional) argument specifies how many numbers to the line (≤ 5); the default is one to a line. If the second argument is ALL, as many values will be printed as have been stored. If less than all are printed, they are the ones at the beginning of the table. The second and third arguments may be numbers or variable names.

Ex. 1 OUTABLE TA,ALL

Prints all of the entries in the table TA.

Ex. 2 OUTABLE TA,N

Prints the first N entries in TA.

Ex. 3 OUTABLE TA,50, K

Prints the first 50 entries in TA, K to a line.

2. Accepting I

INPUT
two l
the s
studen
his me
The ma
380 ch
line

Ex. 1 INPUT

The au
comman
is in

Normal
statem
OTHER,
satisf

Ex. 2 INPUT

If INP
studen

INBELL
from t
Then i
carria
of his
No arg

Ex. 1 INBELL

Ex. 2 INBELL

If the
materi

ies in TA.

ies in TA, K to a line.

2. Accepting Information

INPUT (ACCEPT, ACCEPTEZ). Causes a carriage return, two line feeds, and a question mark to be executed at the student terminal. The computer then waits for the student to enter material. The student indicates that his message is complete by executing a carriage return. The maximum amount of material he can enter is set at 380 characters but this can easily be extended. The line feed key allows multiple lines.

Ex. 1 INPUT

The author can assume, following execution of this command, that student input, up to the carriage return, is in the computer and available for inspection.

Normally, INPUT will be followed by a series of IF-type statements. Such a sequence may be concluded by an OTHER, showing where to go if none of the tests is satisfied.

Ex. 2 INPUT 'NOECHO'

If INPUT is followed by the argument 'NOECHO', the student's message will not be printed at the terminal.

INBELL. Sounds a bell, indicating that input is expected from the student without a carriage return or line feed. Then it waits for the student to enter material. A carriage return by the student is taken to mean the end of his message. A maximum of 380 characters is accepted. No argument is required.

Ex. 1 INBELL

Ex. 2 INBELL 'NOECHO'

If the command has the argument 'NOECHO', the student material will not be printed at the terminal.

3. Analyzing Input

IF (SI). This command has several forms. The basic one calls for two arguments: the first may be a character string or the label of a character string; the second must be the label of another command. If the character string appears anywhere in the student input, the next command is taken from the location indicated by the second argument. Otherwise, the next command in sequence is taken. An alternative form allows the first argument to be a series of character strings or adresses of character strings (separated by commas and enclosed in parentheses). If any one of them appears in the input string, the branch will take place. Another form has a third argument, a number: it is the character position in the input at which the search is to begin.

Ex. 1. IF 'VELOCITY',T34

If the eight character string VELOCITY appears anywhere in the current student input, the next command executed will be the one at T34. Otherwise, the next command in sequence is taken.

Ex. 2 IF ('COW','HORSE','PIG'),T34

If any of the three strings COW, HORSE, or PIG appear in the input, the branch will take place.

Ex. 3 IF 'COW',T34,7

The branch will take place in this case only if the character string 'COW' appears in the input at or after the seventh character the student typed. (This facility is not likely to be needed in most dialoges.)

A typing error in the student's response, or a misspelling, may foil the intention of the 'IF' search. The teacher will often find it advisable to test a part or parts of the desired answer, rather than the whole.

IFONLY. There must be two arguments: the first a character string or the label of a character string, the second a location symbol. If the literal string is identical with the entire input string, the next command is taken from the location indicated by the second argument. Otherwise, the next command is taken in sequence.

Ex. 1 IFONLY 'VELOCITY',T34

If the student typed only the eight characters VELOCITY and a carriage return, the branch to T34 takes place. If he typed more or less than that, it does not.

IFNOT. Thi
IFNOT branc
match betwe
The form of
not possibl

Ex. 1 IFNOT '

A branch to
not write "

Ex. 1 IFNOT

This statem

IFYES. Che
and branche

Ex. 1 IFYES

IFNULL. Ch
no characte
It branches
this is the
the student

Ex. 1 IFNULL

This exampl
type anythi

IFBEFORE.
symbols in
ful match
by an IF.
a label re
of another
the string
word matche

Ex. 1 IF 'EN

E1 IFB

This seque
appears an
tests to s
before the

forms. The basic one
t may be a character
string; the second
nd. If the character
dent input, the next
indicated by the
ext command in sequence
ows the first argument
s or adresses of
mmas and enclosed in
appears in the input
. Another form has a
he character position
s to begin.

CITY appears anywhere
next command executed
, the next command in

HORSE, or PIG appear
place.

s case only if the
the input <u>at</u> <u>or</u> <u>after</u>
typed. (This facility
dialoges.)

sponse, or a misspel-
he 'IF' search. The
ble to test a part or
r than the whole.

nts: the first a
a character string, the
literal string is
tring, the next command
ted by the second argu-
d is taken in sequence.

ht characters VELOCITY
to T34 takes place.
t, it does not.

<u>IFNOT</u>. This command is similar in form to IF. However, IFNOT branches on the opposite condition, i.e., if a match between the argument and the input is <u>not</u> found. The form of IF which allows a set of first arguments is not possible with IFNOT.

```
Ex. 1     IFNOT      'VELOCITY',T34
```

A branch to T34 will take place only if the student did not write "VELOCITY" as part of his statement.

```
Ex. 1     IFNOT      ('COW','HORSE'),T34
```

This statement is <u>illegal</u> and not allowed.

<u>IFYES</u>. Checks for several forms of affirmative reply and branches if one is found.

```
Ex. 1     IFYES      Q3
```

<u>IFNULL</u>. Checks for the condition that the student typed no characters at all, other than the carriage return. It branches to the location specified in the argument if this is the case. The program author can thus check for the student who is not trying.

```
Ex. 1     IFNULL     TRY
```

This example will branch to TRY if the student did not type anything.

<u>IFBEFORE</u>. Takes into account the relative position of symbols in the response. It refers to the last successful match in an IF statement, so it must be branched to by an IF. It has two arguments, a character string (or a label referring to a character string) and the label of another command. It specifies that the match between the string and the input must be found <u>before</u> the last word matched.

```
Ex. 1     IF         'ENERGY',E1

          E1         IFBEFORE 'POTENTIAL',E2
```

This sequence tests first to see if the word ENERGY appears <u>anywhere</u> in the string and, if it does, then tests to see if the word POTENTIAL appears in the string <u>before</u> the word "ENERGY."

IFNEXT (IFAFTER). Takes into account the relative position of symbols in the response. It refers to the last successful match in an IF statement, so must be branched to by an IF. It has three arguments: a character string (or the label of a character string) and two labels of locations in the program. It checks to see if the string appears in the input anywhere after the last match. If so, it branches to the location specified in the second argument and stores all of the characters between the last IF match and the IFNEXT match in the location specified in the third argument.

Ex. 1

```
IF        "VELOCITY',V1
...
V1 IFNEXT      'M/SEC',V2,VEL
```

This sequence will go to V2 if the string "M/SEC" appears after "VELOCITY" in the input. It will also store anything appearing between "VELOCITY" and "M/SEC" in VEL, which must be defined. So "THE VELOCITY IS FOUR M/SEC" as a response will store "IS FOUR" in VEL.

After an unsuccessful IFNEXT, any number of IFNEXT's (or IFBEFORE's) can be used sequentially, provided no successful search is made:

Ex. 2

```
IF        'VELOCITY',V1
...
V1 IFNEXT      'M/SEC',V2, VEL
   IFNEXT      'F/SEC',V2, VEL
```

IFNEXTO (IFAFTERO). Is similar to IFNEXT. However, the character string must exactly match the remaining input string. There is no third argument, as no intervening characters may appear.

Ex. 1

```
IF        'F=',H6
...
H6 IFNEXTO     'M*A',H7
```

This sequence will transfer to H7 if "M*A" is the entire and only string appearing after "F=". Thus, "F=M*A**2" would not make a successful match.

IFKE. Recognizes various forms of kinetic energy and branches if one is found.

Ex. 1

```
IFKE      T73
```

Branches to T73 if the input contains a correct formula for the non-relativistic kinetic energy.

IFPE. Recogn
branches if o

Ex. 1 IFPE P77

Branches to P
for potential

IFFILTH Che
language.

Ex. 1 IFFILTH N

Branches to t
contains any

IFTERMS. And
NOTERMS are u
terms in our
are missing,

Ex. 1 IFTERMS

Could be used

If successful
not, to the

The argument
for each term
student can
the author's

All the patt
so there wil
in the expre
the search i
string, or t
where. The
each argumen

The order in
not matter.
the input an
command does
everything b
paren is con

kes into account the relative
n the response. It refers to the
in an IF statement, so must be
It has three arguments: a char-
label of a character string) and
s in the program. It checks to
ears in the input anywhere after
o, it branches to the location
nd argument and stores all of the
e last IF match and the IFNEXT
specified in the third argument.

,V2,VEL

to V2 if the string "M/SEC"
TY" in the input. It will also
ing between "VELOCITY" and "M/SEC"
defined. So "THE VELOCITY IS
onse will store "IS FOUR" in VEL.

IFNEXT, any number of IFNEXT's
e used sequentially, provided no
made:

,V2, VEL
,V2, VEL

Is similar to IFNEXT. However,
must exactly match the remaining
is no third argument, as no inter-
appear.

H7

ansfer to H7 if "M*A" is the entire
ring after "F=". Thus, "F=M*A**2"
essful match.

ious forms of kinetic energy and
ound.

e input contains a correct formula
stic kinetic energy.

<u>IFPE</u>. Recognizes various forms of potential energy and branches if one is found.

Ex. 1 IFPE P77

Branches to P77 if the input contains a correct formula for potential energy.

<u>IFFILTH</u>. Checks the input string for objectionable language.

Ex. 1 IFFILTH NONO

Branches to the statement labelled NONO if the input contains any of several common swear words.

<u>IFTERMS</u>. And the associated commands ALLWRONG and NOTERMS are useful in determining whether all the terms in our expression are present, or one or more are missing, or have an incorrect sign.

Ex. 1 IFTERMS ('AX**2','A*X+2','A*X**2','ZX+2'),('-BX', '-B*X'),; ('COSTH','COS(TH)'),LABEL

Could be used to check for the expression

$$ax^2 - bx + \cos(TH)$$

If successful, the program would branch to LABEL, if not, to the next sequential instruction.

The argument field of the IFTERMS command includes, for each term expected, all the ways in which the student can write that term correctly (to the best of the author's ability to anticipate this!)

All the patterns for one term are grouped in one argument, so there will be as many arguments as there are terms in the expression, plus a final label to branch to if the search is successful. Each pattern is either a string, or the name of a string that is defined elsewhere. The object of the game is to match one string in each argument to a term in the input, with no leftovers.

The order in which the student enters the terms does not matter. Leading plus signs can be omitted, both in the input and in the IFTERMS command. At present this command does nothing with parenthesized quantities: everything between a left paren and its matching right paren is considered part of the current term.

ALLWRONG. After an unsuccessful IFTERMS, allows the instructor to branch to the sequence appropriate to the mistake or misunderstanding. It must follow directly on the IFTERMS statement.

Ex. 1 ALLWRONG (TERMS,GG)

Tests to see if all the terms expected are missing; if so, it branches to GG.

Ex. 2 ALLWRONG (SIGNS,SS)

Tests to see if all the terms are there, but all with the wrong sign, in which case it branches to SS.

Ex. 3 ALLWRONG (TERMS,G1),(SIGNS,S2)

Transfers to G1 if all the expected terms are missing, but goes to S2 if they are all right except for the sign on each.

NOTERMS. Allows the programmer to test on each or any of the terms separately after an unsuccessful IFTERMS. It also sorts out null strings and syntax errors, if requested, and checks for the case where we find all the expected terms plus extra term(s). NOTERMS must directly follow either IFTERMS or ALLWRONG.

Ex. 1 NOTERMS (MISSING,(3,G1),(2,G2,S2)),(NULL,NN),
(TOOMANY,TTO),(SYNTAX,ISYN)

Will branch to NN if input is just a carriage return, to ISYN if there is a syntax error, to TTO if the expected terms are there, but there are extra ones in the input. If none of these is true, it will look first to see if the term corresponding to argument 3 is missing, and it will branch to G1 if so. If term 3 was matched, it will look next to see if the second term is missing; if it is matched but with incorrect sign, it will transfer to S2; if no correct match for it was found at all, it goes to G2. If none of these conditions is true the next command after NOTERMS is executed.

The options can be given in any order, identified by the key words as shown in the examples. Any option(s) may be omitted.

MISSING is followed by:

1) the ordinal number in the IFTERM command, of the term being considered;
2) the label to transfer to if the term has not been matched;
3) (if present), the label to branch to if it is matched, but with incorrect sign.

essful IFTERMS, allows the
e sequence appropriate to
nding. It must follow
atement.

rms expected are missing;

rms are there, but all with
ase it branches to SS.

IGNS,S2)

expected terms are missing,
all right except for the

ammer to test on each or any
ter an unsuccessful IFTERMS.
ings and syntax errors, if
the case where we find all
tra term(s). NOTERMS must
ERMS or ALLWRONG.

),(2,G2,S2)),(NULL,NN),
(SYNTAX,ISYN)

is just a carriage return,
ax error, to TTO if the
but there are extra ones in
se is true, it will look first
onding to argument 3 is missing,
f so. If term 3 was matched,
f the second term is missing;
incorrect sign, it will transfer
for it was found at all, it
ese conditions is true the next
xecuted.

n any order, identified by the
examples. Any option(s) may

the IFTERM command, of the
;
to if the term has not been

el to branch to if it is
orrect sign.

The programmer experienced in the use of assembly language can do his own checking after an unsuccessful IFTERM, using the stored information.

R1 contains the # of terms expected in the IFTERM;
R2 contains an error code:

1 if input is null string;
2 if syntax error;
4 if all terms were matched, but there are excess terms in the input;
5 if one or more terms were not matched by input terms.

If R2=5, two words store bit information:

#GFLAG contains a 0 bit for each term matched, a 1 bit for each term not matched, in the order given in the IFTERMS command. The remaining word bits are 0.

#SFLAG contains, in the same order, a 1 bit for each term which would have a match but for the sign, 0 for each other term.

TO (OTHER, AUTRE, B, JUMP). The simplest form of this command has a single argument, a statement label, and causes an unconditional branch to that statement. If a second argument is present, it indicates the condition under which such a branch is to take place. This argument is complex and is enclosed in parentheses: it has either two or three parameters: the name of a counter, a relationship (optional), and a number. The relationship may be GE (greater than or equal to), GT (greater than), NE (not equal to), LT (less than), LE (less than or equal to), EQ (equal to); if none is stated, GE is assumed. The branch takes place only if the counter correctly satisfies the specified relationship.

Ex. 1 TO Q5

The next statement to be executed is the one at Q5.

Ex. 2 TO Q7,(CA,LT,5)

Means, branch to Q7 if the counter CA is less than 5; otherwise take the next statement in sequence.

4. Manipulating Strings

NOBLANK. Takes the blanks out of a string, which may be specified in the argument. If no argument is present, the input buffer (#IN) is assumed. Its normal use is after INPUT, when a match requires no blanks; it is particularly valuable in processing formulae or equations, where blanks can appear in random places.

Ex. 1 NOBLANK

Takes the blanks out of the input buffer. Thus, if the student had typed "HORSE MAN SHIP", after this command, the input buffer would contain "HORSEMANSHIP".

Ex. 2 NOBLANK LAST

Will take the blanks out of the string stored at location LAST.

DELETE (REMOVE). Removes part of the input string. It has one argument, a literal string or the label of a literal string, which is to be removed. The argument may be multiple, a series of literals enclosed in parentheses. In this case the first occurence of each string will be removed from the input. The first occurrence of that string is removed from the input.

Ex. 1 DELETE '*'

Removes the first asterisk from the input. If the input does not contain an asterisk, the string is unaltered.

Ex. 2 DELETE ('*',')','(')

Remove the first *, the first), and the first (.

DELETEALL. Removes part of an input string. It has one argument, a literal string or the label of a literal string, indicating the characters that are to be removed. The argument may be a series of literals enclosed in parentheses; all occurrences of each string will be removed from the input.

Ex. 1 DELETEALL ","

Removes all commas from the input string. If these are no commas, the input is unaltered.

Ex. 2 DELETEALL ('A','E','I','O','U')

Removes the vowels a, e, i, o, and u from the input string.

REPLACE
of a sp
Two lit
occurre

Ex. 1 REPLACE

Replaces
time it

SUBALL.
in the
are req
string

Ex. 1 SUBALL

Replaces
there is

ADDAST.
them in
between
order),
followi
exponen
argument

Ex. 1 ADDAST

Converts
A**
this com
A↑2

MOVE (MO
location
from whi
number o
omitted,
moved.
he is mo
the stri
If he sp
he will

For the
language
availabl
machine

Ex. 1 MOVE

Moves th

s out of a string, which may
ent. If no argument is present,
assumed. Its normal use is
requires no blanks; it is
rocessing formulae or equations,
random places.

e input buffer. Thus, if the
IAN SHIP", after this command,
ntain "HORSEMANSHIP".

of the string stored at location

part of the input string. It
l string or the label of a
to be removed. The argument
of literals enclosed in
e the first occurence of each
m the input. The first
is removed from the input.

k from the input. If the input
isk, the string is unaltered.

irst), and the first (.

of an input string. It has
tring or the label of a literal
aracters that are to be removed.
ies of literals enclosed in
ces of each string will be

e input string. If these are
altered.

','O','U')

i, o, and u from the input

<u>REPLACE (SUB, SUB:FOR)</u>. Replaces the first occurrence of a specified string in the input with a second string. Two literal strings are required as arguments: the first occurrence of the first string is replaced by the second.

Ex. 1 REPLACE 'TWO','2'

Replaces the character string "TWO" with "2" the first time it appears in the input.

<u>SUBALL</u>. Replaces each occurrence of a specified string in the input with a second string. Two literal strings are required as arguments: each occurrence of the first string in the input is replaced by the second.

Ex. 1 SUBALL '**','↑'

Replaces each double asterisk with the up-arrow. If there is no **, the string is unaltered.

<u>ADDAST</u>. Takes formula input by students and transforms them into a BASIC-like form. It inserts asterisks between letters, or between numbers and letters (in either order), and between a number or letter and the parentheses following or preceding it. It replaces the FORTRAN exponentiation "**" with "↑". It removes blanks. No argument is required.

Ex. 1 ADDAST

Converts an input formula. If the student had typed,

```
    A**2 + 2AB + B**2
```

this command would convert it to

```
    A↑2+2*A*B+B↑2
```

<u>MOVE (MOVEZ)</u>. Moves all or part of a string from one location to another. The arguments specify the strings from which and to which the move is to take place; the number of characters to be moved (if this parameter is omitted, it is assumed that all of the string will be moved. The user must take care that the location to which he is moving the string is defined large enough to contain the string moved, else he may overwrite other material. If he specifies more characters than the string contains, he will move garbage along with the string he wants.

For the more advanced programmer who uses some assembly language in his programs, special forms of MOVE are available in which some parameters can be stored in machine registers. See examples below.

Ex. 1 MOVE A,B

Moves the entire string at A to B. A is unchanged.

Ex. 2 MOVE A,B,40

Moves the first 40 characters at A to B. If A does not contain 40 characters, garbage will be moved with it.

Ex. 3 MOVE SAY,WHEN,*1

Moves the initial N characters of the string SAY, where N is the number in register 1, to string storage in WHEN.

Ex. 4 MOVE (HOW,3),MUCH,*2

Moves K characters of string HOW, starting with character number 3, to MUCH, where K is the number in register 2.

Ex. 5 MOVE (HOW,*3),MUCH,*2

Moves K characters of string HOW, starting with character number J, to string location MUCH. K is the value in register 2 and J is the value in register 3.

<u>PTRMOVE</u>. Was designed for moving strings whose location is stored in a known address. The instruction is of the form:

```
PTRMOVE    *A,*B
```

where the address of the string to be moved is stored in A, and the address of the new location is stored in B.

Variations of this are the use of indexings:

```
PTRMOVE    (*A,1),*B
```

moves the string whose address is in the nth word of A, where n is the value stored in register 1. This operation may be applied to the 2nd argument also.

In certain cases the asterisk may be left off either or both arguments.

```
PTRMOVE    A,(*B,1)
```

In this case it is assumed that A is the name of the string to be moved. By leaving off both asterisks, the operation becomes the same as MOVE.

<u>APPEND</u>.
modifying
string re
string to
to be mov
parameter
append mo
has room

Ex. 1 APPEND

Adds stri
count of

haracters at A to B. If A does
acters, garbage will be moved with

characters of the string SAY, where
egister 1, to string storage in WHEN.

CH,*2

of string HOW, starting with character
where K is the number in register 2.

UCH,*2

of string HOW, starting with character
location MUCH. K is the value in
the value in register 3.

ned for moving strings whose location
n address. The instruction is of

,*B

f the string to be moved is stored in
of the new location is stored in B.

are the use of indexings:

A,1),*B

ose address is in the nth word of A,
e stored in register 1. This operation
he 2nd argument also.

e asterisk may be left off either or

(*B,1)

assumed that A is the name of the
By leaving off both asterisks, the
he same as MOVE.

APPEND. Concatenates one string on to the end of another, modifying the character count appropriately. The second string remains unchanged. The arguments specify the string to which and the string from which characters are to be moved. Note that this is the opposite order of the parameters in MOVE. The user should be careful not to append more characters than the defined string location has room for.

Ex. 1 APPEND A,B

Adds string B to the end of string A and modifies the count of string A to reflect A's new length.

5. Manipulating Numbers

NUMBER. Examines a character string to see if it constitutes a recognizable number and, if so, converts to floating point form and stores the number. The first argument is the location in which the number is to be stored; the second argument is the location to go to if the string is not a recognizable number; the third argument, if present, is the location of the string. If there is no third argument, the input buffer #IN is assumed.

Ex. 1 NUMBER TIME,NOGOOD

Examines the input buffer and either stores the converted number in TIME or branches to NOGOOD.

Ex. 2 NUMBER TIME,NOGOOD,NSTRING

Does the same for a string in NSTRING.

A "recognizable number' in this and other commands testing for numbers is defined as being of the following form:

[±]XX[.]XX[E[±]X[X]]

where the brackets indicate optional characters and there can be any number of digits X in the part of the number preceding the exponent.

SCAN. Separates a string into three parts: that part containing a number, the part before it, and the part after. Either four or five arguments are required: the location at which to store the number; the location for the characters before the number; the location for the characters after the number; an error location if no number is found; the string location (if omitted, input buffer assumed). All strings must be defined by the user. If there is no number, a branch to the error location occurs and zero counts are stored in the three specified string locations. All blanks are removed from the string scanned, whether or not the operation is successful.

Ex. 1 SCAN NUMST,STBEF,STAFT,ERR

Scans the input buffer for a string of characters representing a number. That string is stored in NUMST; the characters which preceded it in STBEF; the characters which followed it in STAFT. If no number is found, NUMST,STBEF,STAFT are given zero counts and a branch to ERR occurs.

string to see if it
ber and, if so, converts
res the number. The first
ich the number is to be
the location to go to
able number; the third
ocation of the string. If
e input buffer #IN is

either stores the converted
NOGOOD.

NSTRING.

s and other commands testing
g of the following form:

tional characters and there
in the part of the number

three parts: that part
before it, and the part
guments are required: the
number; the location for
er; the location for the
n error location if no
cation (if omitted, input
must be defined by the
a branch to the error
s are stored in the three
ll blanks are removed from
not the operation is

string of characters
string is stored in NUMST;
it in STBEF; the characters
f no number is found,
ero counts and a branch to

SCAN#. Performs all the functions of SCAN but also converts the number into floating point form for use in computation.

Ex. 1 SCAN# NUMST,STBEF,STAFT,ERR

Stores the string preceding the number in STBEF, the string following the number in STAFT and the number, converted to floating point form, in NUMST.

IFNUMEX. Tests the input string (or any other string) to see if it is a number (only). The first argument is the location to which to branch if the string is exclusively a number; the second argument (if present) is the location of the string to be tested. If absent, input buffer is assumed.

Ex. 1 IFNUMEX NEXT

Branches to NEXT if the input buffer contains only a number.

IFNOTNUM. Is the reverse form of IFNUMEX. It tests the input string (or any other string) to see if it is a number (only). The first argument is the location to which to branch if the string is not exclusively a number; the second argument (if present) is the location of the string to be tested. If absent, input buffer is assumed.

Ex. 1 IFNOTNUM NEXT

Branches to NEXT if the input buffer is not exclusively a number.

AROUND. Tests the range of a floating point number. There are four arguments; the number, the central value, the deviation from this value allowed, and the successful branch point.

Ex. 1 AROUND N,S,E,GOTO

IF S-E<N<S+E, the program will branch to GOTO; if not, the next instruction in sequence will be taken. S and E are the locations of floating point numbers.

Ex. 2 AROUND K,FS'0.5',FS'0.02',BRANCH

IF .48<K<.52, it branches to BRANCH, else takes the next instruction. "FS" here indicates a floating point number.

BETWEEN. Tests the size of a number. There are four arguments: the number to be tested, the lower bound, the upper bound, and a branch location. The bounds can be either locations where the bounds are stored or actual floating point numbers.

Ex. 1 BETWEEN N,BOTTOM,TOP,GOTO

If the number at N is between the values of BOTTOM and TOP, inclusive, it branches to GOTO; if not, the next instruction in sequence is taken.

Ex. 2 BETWEEN N,FS'12.5',FS'12.8',T749

If the number at N is between 12.5 and 12.8, it branches to T749. Note the format of floating point constants; see the Metasymbol manual for further details.

RANDOM. Generates and stores random numbers. The first argument is the location at which it is to be stored. The second and third arguments (optional) indicate the range. If they are omitted, the number will be between 0 and 1. A sequence of random numbers generated by a series of calls is unique: no other runs of the program will generate the same sequence.

Ex. 1 RANDOM X,A,B

Generates a random number between A and B and stores it in X.

Ex. 2 RANDOM Y

Generates a random number between 0 and 1 and stores it in Y.

Ex. 3 RANDOM A,FS'0',FS'50.0'

Generates a random number between 0 and 50 and stores it in A. ("FS" indicates a floating point number.)

RANDOMR. Is like RANDOM, except that the sequence of numbers generated is repeatable: that is, every run of the program will generate the same sequence of pseudo-random numbers.

a number. There are four
e tested, the lower bound,
ch location. The bounds
re the bounds are stored or
rs.

O
.

en the values of BOTTOM and
to GOTO; if not, the next
taken.

2.8',T749

en 12.5 and 12.8, it branches
f floating point constants;
or further details.

es random numbers. The first
which it is to be stored.
nts (optional) indicate the
, the number will be between
dom numbers generated by a
no other runs of the program
ence.

etween A and B and stores it

etween 0 and 1 and stores it

etween 0 and 50 and stores it
oating point number.)

xcept that the sequence of
able: that is, every run of
he same sequence of pseudo-

6. Manipulating Counters

BUMP (INCREASE, AUGMENT, ADD1). Increases the value of a counter (or counters) by 1. The argument is the name of the counter (or counters, separated by commas and enclosed in parentheses). The maximum value of a counter is 255.

Ex. 1 BUMP C1

Ex. 2 BUMP (C2,C23,A)

DECREASE (SUB1). Is used to decrease the value of counters by 1. The argument specifies the name of the counter to be bumped, or the counters, if more than one, separated by commas and enclosed in parentheses.

Ex. 1 DECREASE C1

Ex. 2 DECREASE (C2,C23,A)

The minimum value for a counter is 0.

RESET (ZERO). Stores a new value in specified counters. The first argument is the name of the counter or counters; the second is the value to which the counter is to be set. If the second argument is missing, zero is assumed to be the value.

Ex. 1 RESET (AB,C2,Q17)

Gives the three counters the value of zero.

Ex. 2 RESET Q17,4

Gives the counter Q17 the value 4.

ADDCOUNT. Sums two or more counters and stores the result in one of them. The first argument is the name of the counter in which the sum is to be stored. The second argument is the additional counter (or counters, separated by commas and enclosed in parentheses).

Ex. 1 ADDCOUNT S,A

Adds counter A to S and stores the sum in S.

Ex. 2 ADDCOUNT S,(A,B,R)

Adds counters S, A, B, and R and stores the sum in S.

CTARITH. Enables the user to add, subtract, multiply or divide counters. The operation is specified in the second argument field. The fourth argument is the branch point in case of error; all counters remain in original form. Error conditions are

overflow -- value > 255
underflow -- value < 0
division by 0

```
Ex. 1    CTARITH    A,ADD,B,C
                    ADD A TO B LEAVE IN A

Ex. 2    CTARITH    A,SUB,B,C
                    SUB B FROM A LEAVE IN A

Ex. 3    CTARITH    A,MULT,B,C
                    MULT A BY B LEAVE IN A

Ex. 4    CTARITH    A,DIV,B,C
                    DIV A BY B, TRUNCATE AND LEAVE IN A
```

CTWRITE. Outputs a CRLF and then the value of any counter in decimal form.

```
Ex. 1    CTWRITE    T2
```

If T2 has the value 5, this generates a carriage return and line feed, then the number 5.

CTOUT. Output just the counter value (no CFLF).

```
Ex. 1    CTOUT      A
```

If the counter A is 2, this prints 2 on the current line.

SWITCH. Is a command for testing the value of a counter and branching to one of several locations, depending on its value. The first argument is the name of a counter. The second argument is a set of statement labels (separated by commas and enclosed in parentheses) to which to branch on sequential values of the counter, starting with zero. If the value of the counter is greater than the number of branch points supplied, the SWITCH is ignored and the next command is executed.

to add, subtract, multiply or
ation is specified in the
e fourth argument is the branch
l counters remain in original
e

55
0

VE IN A

EAVE IN A

AVE IN A

UNCATE AND LEAVE IN A

nd then the value of any

is generates a carriage return
mber 5.

unter value (no CFLF).

is prints 2 on the current line.

testing the value of a counter
everal locations, depending on
ment is the name of a counter.
set of statement labels
enclosed in parentheses) to
ial values of the counter,
he value of the counter is
f branch points supplied, the
next command is executed.

For specifying a number of branches, SWITCH is more efficient in execution than a series of TO's although functionally equivalent.

Ex. 1 SWITCH A,(A0,A1,A2,A3,A4)

Branches to A0 if A is zero, A1 if A is 1, A2 if A is 2, and so on, but goes to the next command in sequence if A is 5 or larger.

7. Constant and Parameter Storage

DEFINE (DEFINEZ). Reserves space for the storage of strings of characters, including the character count, and defines the label which will be used to refer to it. The first argument is the label, the second is the number of characters. If the second is omitted, 16 characters are assumed. The user can use the same space for different things in different parts of his program. It is his responsibility to be sure a string area is large enough to contain the string moved into it or appended to it. There is no limit to the length of a string or string area but if the string is to be used as a message it should be limited to what can be printed on one line (70 characters).

Ex. 1 DEFINE STR1

Reserves storage for 16 characters, to be referred to as "STR1".

Ex. 2 DEFINE STR2,70

Reserves storage for 70 characters, to be referred to as "STR2".

COUNTER (COMPTEUR, C). Defines a label as referring to a counter and reserves space for that counter. The first argument is the label or a set of labels for several counters, separated by commas and enclosed in parentheses. The second argument is the initial value the counter(s) is to have. If the second argument is not present, the counter(s) will have an initial value of zero. Currently, 32 counters are allowed in the program; the maximum value for a counter is 255. Any number of COUNTER statements may appear in a program (up to 32), but no counter should appear in more than one COUNTER statement as this is a multiple definition of a label.

Ex. 1 COUNTER AQ8

Establishes a counter to be called AQ8, with an initial value of zero.

Ex. 2 COUNTER B,5

Establishes a counter, B, with an initial value of 5.

Ex. 3 COUNTER (C,D,E,F),7

Establishes counters C, D, E, and F, each with initial value of 7.

ng

erves space for the storage of
including the character count,
which will be used to refer to
is the label, the second is the
If the second is omitted, 16
The user can use the same
ngs in different parts of his
sponsibility to be sure a string
contain the string moved into
There is no limit to the length
rea but if the string is to be
ould be limited to what can be
characters).

characters, to be referred to

characters, to be referred to

Defines a label as referring to
space for that counter. The
label or a set of labels for
rated by commas and enclosed in
nd argument is the initial value
ave. If the second argument is
er(s) will have an initial value
2 counters are allowed in the
alue for a counter is 255. Any
ements may appear in a program
nter should appear in more than
as this is a multiple definition

to be called AQ8, with an initial

B, with an initial value of 5.

7

, D, E, and F, each with initial

Counters can have any name (within the limits on label names) as long as it is not used for any other purpose. One convenient procedure is to use a name related to the label of the WRITE statement where the counter is first used; thus, if the WRITE statement is labelled T32, the counter might be labelled T32C. Or, the counter name might indicate the function of the counter. But the user does not have to follow any such naming suggestions.

SET. A metasymbol command enables the user to define (at assembly time), a label or labels by assigning to each the attributes of the list in the argument field. It can be used to supply synonyms for counter names, for instance (this may be useful for mnemonic purposes.) It can also be used to give specific arithmetic values to labels. Labels have the values assigned until they are redefined by another SET statement.

Ex. 1

```
ABC     SET     DEF
A1      SET     23
A2      SET     A1*2+3
```

In these statements, ABC is given as a synonym for DEF (which must be defined elsewhere; A1 is the number 23 and A2 is defined in terms of A1. Labels originally defined by an EQU or by a COUNTER statement may not be redefined with a SET.

TEXT#. Defines a character string, specifying the name which will be used to refer to it. The label field contains the name of the string; the argument field contains a literal string.

Ex. 1

```
AZ    TEXT#     'VELOCITY'
```

Stores the characters "VELOCITY" at a location identified as AZ. Thus a statement WRITE AZ will produce the word VELOCITY at the student terminal.

Strings are stored internally in the conversational system in a fashion different than in supplied Sigma-7 software. The full first word of this string contains the number of characters in the string; the characters begin with the first (left-most) byte of the second word. The command enters strings in this fashion. This change was made because in some instances it is advisable to work with strings of more than 256 characters, the maximum which can be stored with the TEXTC command in Metasymbol. All of the procedures and subroutines in the system assume string storage as just outlined.

STRING. Is similar to TEXT#, but it has a branch around the text-string itself. So STRING can be "executed"(it does nothing), while TEXT# cannot. The beginner who is uncertain of this distinction should use STRING. A string which needs to written out many times should be stored this way.

Ex. 1 GREET STRING 'GOOD DAY!'

Stores the string 'GOOD DAY!' under the name GREET.

DEFNUM. Reserves storage space for individual floating point numbers and assigns labels to the storage. The argument is the label or labels (enclosed in parentheses, separated by commas).

Ex. 1 DEFNUM WEIGHT

Reserves a single storage space for a variable WEIGHT.

Ex. 2 DEFNUM (X,Y,Z)

Reserves storage for three variables.

Used with one argument, this directive only reserves storage and does not establish any initial value. Assigning values is the user's responsibility.

An optional second argument can be used to store an initial value for each label defined in the first argument. The second argument is one number, or one symbol only.

Ex. 3 DEFNUM (X,Y,Z), FS'3.5'

Will define locations X, Y and Z and place a floating short 3.5 in each of them.

Ex. 4 DEFNUM K, R

Will define location K, and store in it the floating point number which is now in location R.

#, but it has a branch around
STRING can be "executed"(it
cannot. The beginner who is
on should use STRING. A
en out many times should be

DAY!'

!' under the name GREET.

pace for individual floating
abels to the storage. The
bels (enclosed in parentheses,

pace for a variable WEIGHT.

variables.

s directive only reserves
ish any initial value.
r's responsibility.

can be used to store an
l defined in the first
ent is one number, or one

and S and place a floating

store in it the floating
n location R.

STORENUM. STORENUM has two arguments. The first is a location previously defined by DEFNUM, and the second is a floating short number which is then assigned the first argument.

Ex. 1 STORENUM A,FS'1'

Places a floating short one in A which is previously defined.

DEFCOMP. DEFCOMP works in the same manner as DEFNUM except that for each label it reserves a double word. It also may assign a value at time of definition. This is done with a complex second argument which contains two floating short numbers. The reserved number may then be handled with doubleword commands.

Ex. 1 DEFCOMP A,(FS'1',FS'2')

Will reserve two words addressed by A with a floating short 1 in the first and a floating short 2 in the second.

STORECOMP. STORECOMP will store two floating short numbers in a previously defined complex number.

Ex. 1 STORECOMP A,(FS'1',FS'2')

Would store a floating short 1 and 2 in the doubleword defined as A.

DEFTABLE. Reserves storage space for tables or linear arrays of floating point numbers. The first argument is the label to be assigned, the second is the number of words in the table.

Ex. 1 DEFTABLE TIMETAB,100

Reserves 100 words for a table which will be referred to as TIMETAB. Numbers can be stored easily in tables like this using the command STACK.

STACK. Stores numbers into a table or linear array (which must have been defined using the DEFTABLE directive). The first parameter is the location of the number; the second is the name of the table; the third, if given, is the branch point for overflow. (If no third argument is given, table overflow is marked by a warning printout: "Table overflow, value not stored," and the next sequential instruction is executed. This is not generally advised. STACK is useful in simulations for storing student measurements.

Ex. 1 STACK TIME,TIMETAB

This will store the current value of the number TIME into the next available space in the table TIMETAB; i.e., if there are 100 words reserved for TIMETAB, and 16 have already been filled, TIME will be stored in the 17th.

manner as DEFNUM
es a double word.
f definition. This
nt which contains
erved number may
nds.

A with a floating
short 2 in the

floating short
lex number.

in the doubleword

r tables or linear
he first argument
ond is the number

will be referred
d easily in tables

or linear array
the DEFTABLE
the location of the
e table; the third,
erflow. (If no
low is marked by a
value not stored,"
is executed. This
useful in simulations

the number TIME
table TIMETAB; i.e.,
TIMETAB, and 16 have
ored in the 17th.

8. Other Commands

SYSTEM DIALOG. Directs the assembler to select the file DIALOG containing the commands that are to be valid during this assembly. This must be the first statement in the program.

NAME
START

Either NAME or START must follow the SYSTEM DIALOG command at the beginning of the program.

START. Initializes the flow of instructions when execution begins.

NAME 'AJAK'. Does the work of START, but in addition uses the (4 or less) characters in the argument to name the response file on which the students' responses will be saved if any of the SAVE commands are used. In this case the file name would be 'RESAJAK'. It also stores these characters as part of the students' ID on the name file, which keeps records of starts (and restarts, if ENTRY is used).

END. This indicates to Metasymbol that it is the last statement to be processed. See Sec. 1-2 for discussion of END DIALOGUE and END without an argument, or with a different argument.

STOP (HALT). Indicates that this is the last instruction in the program which will be executed: i.e., when the student uses the sequence, he is done at this point, and control is returned to the executive. It also erases from the name file the record containing this student's ID. It is not necessarily the last statement in the program. Nor need it be unique: there may be several exits from the program.

FINALE (EPILOGUE, EPILOG). Indicates that this is the last instruction in the program which will be executed. When the student reaches this point, he is asked to type a comment about the sequence. This comment is saved on disk. Control is then returned to the executive. Use of this instruction is optional; some authors prefer to do this themselves, or not do it at all.

ENTRY. Is the command which permits restart. It does not need to be used. If it is used, the student who does not finish a conversation in one sitting can restart at some place other than the beginning. The command Ex. 3
ENTRY should be used at all locations at which the teacher wishes to allow a restart. Normally, it should be just before a WRITE command, so that the student will not be restarted at an input. Restart occurs at the last executed ENTRY. Ex. 4

If no ENTRY is used in the program, the student begins directly with the user program. If one or more ENTRY commands are used, he is first asked to type an ID; this identification is used for restarting. Further, if the program uses any ENTRY commands, the student is reminded of his ID after he types STOP at any input.

In a program using restart, if the student uses a previous identification, he is asked whether he has used the dialogue before. If so, he is restarted at the last entry point executed when he first ran the program.

SAVE (KEEP, INFO). Causes information to be stored on a disk file (while the program is running) for later study. It has two forms: to save student responses and to save counters. The form to save responses has one argument,
a character string which will serve as the name of the Ex. 1
record as it is stored on disk. When SAVE is encountered in running the program, the contents of the input buffer, the date, the time, and the name are saved. The name should be no longer than 40 characters. One possibility is to use the label of the preceding WRITE statement. The SAVE command for preserving the values of the counters has three arguments: the first must be COUNTERS; the second is the name of the counters (separated by commas and enclosed by parentheses) or ALL; the third is a character string to serve as a name. If the third argument is omitted, the name "COUNTERS" will be used.

Ex. 1 SAVE 'RESPONSE 1'

Saves the input buffer, time, date, and the name Ex. 2
"RESPONSE 1" on disk.

Ex. 2 SAVE COUNTERS,ALL,'KEPLER'

Saves all of the defined counters under the name "KEPLER".

its restart. It does
ed, the student who
one sitting can restart
nning. The command
ions at which the
. Normally, it should
o that the student will
start occurs at the

m, the student begins
If one or more ENTRY
ked to type an ID;
estarting. Further,
mands, the student is
STOP at any input.

e student uses a
ked whether he has used
is restarted at the last
t ran the program.

ation to be stored on a
running) for later study.
t responses and to save
nses has one argument,
ve as the name of the
When SAVE is encountered
nts of the input buffer,
are saved. The name
cters. One possibility
ding WRITE statement.
the values of the counters
must be COUNTERS; the
rs (separated by commas
ALL; the third is a
ame. If the third
OUNTERS" will be used.

te, and the name

s under the name

Ex. 3 SAVE COUNTERS,(C1,C2,C3),'K'

Saves the three listed counters under the name "K".

Ex. 4 SAVE COUNTERS,K2

Saves only the counter K2 and assigns the record the name "COUNTERS".

SAVEID. Is identical in function to SAVE except that the student ID is preserved as part of the disk record.

FORTRAN. Allows the user to introduce FORTRAN subroutines into his dialogue. Any number of FORTRAN subroutines can be called any number of times within a dialogue program subject only to the limitations of space. All FORTRAN facilities in XDS FORTRAN IV are available to the user. The subroutine itself must be compiled using the XDS FORTRAN IV compiler (not IV-H) and is loaded along with the rest of the program.

The argument of the command is the name of the FORTRAN subroutine together with (in parentheses) the arguments for the subroutine.

Ex. 1 FORTRAN POLLY,(X,Y,Z)

The default assumption is that the subroutine arguments are real variables; the user can specify if he wants the variables to be integers, complex numbers, etc., according to the following table:

1	Integer
2	Real
4	Real double precision
8	Complex
10	Double complex
20	Logical
3F	Any argument type

Ex. 2 FORTRAN NATINV,((I,1),(J,1),(Z,8))

Calls a subroutine in which I and J are integers and Z is complex.

HERE (MARK). Simply identifies a location in the program.

LOAD (TRANSFER). Brings into core the program segment whose binary name (in single quotes) is the first argument. If the second argument is present, a branch is made to that label. Please read Section 3.7 on OVERLAYING for a more complete discussion.

Ex. 1 LOAD 'CONIBO',C5

Brings into core the binary file named CONIBO, and branches to C5.

Ex. 2 TRANSFER 'BR3'

Brings into core the binary file whose name is BR3. Presumably the programmer will later transfer to the instructions in BR3 by a TO command or its equivalent.

AUDIO. Works like a WRITE command, except that the message is delivered in an audible fashion using the RAAD audio response device, (William M. Brobeck and Associates), rather than through the teletype.

Ex. 1 AUDIO VOICEI

VOICEI is the location in which information is stored identifying the audio record.

TALK. To indicate the audio record, five pieces of information must be given, for the RAAD device. These are given in order, as a set of hexadecimal digits by the following command.

VOICEI TALK X'(digits)'

Thus TALK supplies the information about the location of the audio record. It must be labeled and referenced by the audio command that actually produces the talking. VOICEI is a label, starting in column 1.

ANSWER. Works like INPUT, except that it checks the student input for REPEAT, repeats the last audio message, and continues from there. Normally, it should be used only after AUDIO.

GOOD. GOOD will randomly choose one of ten statements equivalent to "correct". It may be used without any argument.

Ex. 1 GOOD

Ex. 2

core the program segment
quotes) is the first
ment is present, a branch
e read Section 3.7 on
e discussion.

ile named CONIBO, and

ile whose name is BR3.
l later transfer to the
ommand or its equivalent.

mmand, except that the
dible fashion using the
William M. Brobeck and
ugh the teletype.

ch information is stored

record, five pieces of
r the RAAD device. These
of hexadecimal digits by

)'

ation about the location
be labeled and referenced
ually produces the talking.
n column 1.

cept that it checks the
eats the last audio
here. Normally, it should

ose one of ten statements
may be used without any

In which case the next instruction will be executed after the GOOD response is printed.

It may also be called with an argument which is the address of a location.

Ex. 2 GOOD OUT

In this case, after printing the message the program will branch to OUT.

AGAIN. Will randomly choose one of 10 statements equivalent to 'try again'. It is used with or without an argument, as in the command GOOD.

GREETING. Displays a greeting to the student appropriate to the time of day:

Good Morning,

Good Afternoon; or

Good Evening

CHAPTER 3
GETTING IT ON THE MACHINE

If the reader has had no previous experience using the BTM timesharing system, he will need some assistance in learning how to load, debug, and run his program. Probably the most effective method is to ask an experienced Sigma 7 user to spend a little time with you until you know the ropes; after that, you can refer to the XDS manuals for additional information. However, it may be helpful to have some material in this convenient location. We have by no means attempted to explain the full power of the systems described; only enough information is provided to get the beginner going.

In what follows, it is assumed that the program is to be typed at a terminal and stored on disk; it will be modified and corrected, assembled, tested, and finally made available for student use. Before you begin, you will need a legal account number; check with your computer center on this. You will need to know the account containing the dialogue system and libraries as well. (At Irvine, it is B9999.)

1. Using the terminal

To sign on, you must press the "Break" key. The system will announce itself and ask you to identify yourself with name and account number. Once this is accomplished, the system will type an exclamation point, which means that it is waiting for you to

erience using the BTM
assistance in learning how
Probably the most effective
7 user to spend a little
s; after that, you can refer
ormation. However, it may
this convenient location. We
the full power of the systems
provided to get the beginner

the program is to be typed at
ll be modified and corrected,
available for student use.
gal account number; check with
ill need to know the account
ibraries as well. (At Irvine,

k" key. The system will
ify yourself with name and
lished, the system will type
at it is waiting for you to

tell it what you would like it to do next. Here are some of the functions of the monitor which you will want to be able to use. In these examples, messages output by the monitor are underlined. Note especially that, in calling for any of these functions, the user types only the first two characters; BTM finishes the word.

> !LOGIN: (this asks the user to type his name and account number. When you have done this, press carriage return. If you are acceptable to the system, it will type an exclamation point and wait for your command.)
>
> !TABS (type the numbers of the character positions where you would like to have tab stops, separated by commas. Carriage return when you are through.)
>
> !EDIT (calls for the text editing program which you will use to input your program. More below on this.)
>
> !BYE (indicates that you are finished and wish to sign off.)

There are some special typing conventions which the user should be aware of:

> Return to executive: when the user is in some system or subsystem, he may wish to return control to the executive. He does this by pressing the "escape" key twice -- perhaps several times.
>
> Backspace: "escape" and "rubout", will cause an effective backspace in the information being stored in the machine. It may not cause an optical backspace in the material you see on the terminal, however.
>
> Erase: "escape" "X" causes the entire present line to be deleted from the machine. Again, it will not necessarily be removed from the terminal.
>
> Retype: "escape" and "R" cause the complete statement to be retyped. If there have been backspaces and erasures, it is sometimes nice to be able to see exactly what you have done. The machine waits for further input.

<u>Tab</u>: "escape" and "I" cause spaces up to the next tab stop. Tabs must have been set previously at the executive level.

<u>Check status</u>: "escape" and "Q" cause the system to type "!!". This is useful (after a long pause, for example) for checking whether the system is still alive.

2. Using EDIT

The EDIT system allows the user to create, modify, and list disk-resident files. This is one way to enter programs. (Cards are also a possibility.) Once the user is satisfied that his program is complete, he can then assign it as an input file for Metasymbol.

EDIT announces itself with an asterisk and waits for the user to specify one of its commands. All of them will not be discussed here; the BTM Users Manual gives a user-oriented description of each. You should be able to get a good start with the pieces described here.

*BUILD fid (i.e., EDIT types the asterisk, you type "BUILD" and then some file identification of your choice, then carriage return.) When this command is accepted, a new file is created on disk with the name you have specified. EDIT then types a line number (1.000) and waits for you to fill the line. A carriage return, as usual, terminates the line and another line number is typed. If you return the carriage without typing any characters, it is assumed that you are done with BUILD and want to call another EDIT function -- it types another asterisk. Because of the oddities of the BTM system, it is wise to terminate BUILD in this way before you get too far in your typing so as to establish your file on disk, and then add to it using the IN command.

*END Returns control to the executive -- which types an exclamation point to let you know it is there.

ause spaces up to the next tab
een set previously at the

and "Q" cause the system to
ul (after a long pause, for
ether the system is still alive.

o create, modify, and list disk-

to enter programs. (Cards are

er is satisfied that his program

t as an input file for Metasymbol.

erisk and waits for the user to

of them will not be discussed

a user-oriented description of

a good start with the pieces

types the asterisk, you type
ile identification of your
eturn.) When this command is
created on disk with the name
IT then types a line number
ou to fill the line. A carriage
nates the line and another line
u return the carriage without
it is assumed that you are done
call another EDIT function -- it
Because of the oddities of the
to terminate BUILD in this way
in your typing so as to establish
then add to it using the IN

o the executive -- which types an
t you know it is there.

*DELETE fid This allows you to remove the file from disk. It effectively destroys the file and all references to it.

*EDIT fid This allows you to edit a text stored on disk. All this command does is indicate that you wish to edit a particular file, which you identify by name. Following it, you must specify what you want to do. Only a few of the possibilities will be described here: IN, DE, TY, SE.

*DE n-m Deletes records n through m. Do not confuse DE with DELETE which removes the entire file.

*TY n-m Types records n through m. If m is omitted, only n will be typed. If you wish to type to the end of the file and do not know the number of the last record, simply use a very large number for m.

*SE n;/str1/S/str2/;TY SE is a powerful command with many variations. This one is particularly useful: In line n, substitute str2 for str1 (first occurance only) and type the new line. Follow the indicated punctuation carefully.

3. Using Metasymbol

Metasymbol programs must be assembled as a batch job -- in BTM they cannot be assembled on-line with direct feedback on the terminal. The control information and the instructions for the assembly, however, can be submitted from a terminal as well as from cards. In either case, the file with the course material will be input to the assembly program.

The following suggested possibilities for assembly assumes BTM version E00; later versions may have different characteristics. Let us assume that the conversational file stored on disk is called COURSE and that we intend to give the binary output of

the assembler the name COURSEBO. The following 'cards' will perform the assembly (first character in column 1):

```
!JOB          (accounting information--inquire
               locally for details)
!LIMIT        (TIME,5)
!ASSIGN       M:SI,(FILE,COURSE)
!ASSIGN       M:BO,(FILE,COURSEBO)
!METASYM      LS,SI,BO,AC(B9999)
```

The JOB card contains accounting information; details may vary from installation to installation; someone familiar with your computer set-up will be able to help you here. The second card sets a limit on the amount of time to be used in this particular job. Setting a five minute limit simply protects you from the chance of some error which would cause your job to run on endlessly -- and your account to be billed accordingly. The ASSIGN cards specify files related to your program. The system input (:SI) is to be an existing disk file (FILE) called COURSE. The binary output of the assembly (BO) is to be a file called COURSEBO. (There are other functions which the ASSIGN directive will perform; details will be found in the BPM and BTM manuals.) Pay particular attention to the punctuation of these statements; they have been the despair of more than one amateur typist.

The final card of this set indicates that the system program you will be using is Metasymbol (METASYM). The information in the

BO. The following 'cards' will
haracter in column 1):

(accounting information--inquire
locally for details)
(TIME,5)
M:SI,(FILE,COURSE)
M:BO,(FILE,COURSEBO)
LS,SI,BO,AC(B9999)

ing information; details may vary
ation; someone familiar with your
to help you here. The second card
f time to be used in this particular
limit simply protects you from the
ould cause your job to run on
t to be billed accordingly. The ASSIGN
to your program. The system input
isk file (FILE) called COURSE. The
y (BO) is to be a file called COURSEBO.
hich the ASSIGN directive will perform;
BPM and BTM manuals.) Pay particular
of these statements; they have been
amateur typist.

ndicates that the system program you
(METASYM). The information in the

argument field is to specify what precisely you wish the assembly program to do. Three of these arguments are required in our situation:

SI specifies source input

BO specifies binary output

AC specifies that the dialogue procedures are to be found in a file in account B9999. (At installations other than Irvine, this number may be different.)

Other arguments are optional:

LS requests a listing of the source program

LO requests a listing of the output -- assembly language and machine language code generated.

CN requests a concordance. The METASYM card must in this case be followed by one or more concordance cards (see Metasymbol Manual). The last card must contain (columns 1-4) .END.

SD specifies symbolic debugging facilities: special dictionaries are prepared and saved so that the DELTA debugging program can be used. (Cannot be used in the "final" version.)

To enter the batch processing system from the terminal, type BP at the executive level (i.e., after a ! prompt character). Y is the correct response to "INSERT JOB?"; no carriage return is needed. Then the above four lines can be typed. A blank line terminates input and the user can reply 'N' to the "EDIT?" question. A terminal message indicates that the job has been inserted. From the standpoint of the computer, this job is just the same as if it had come in from cards. You can pick up the (line printer) output from the computer center; if you are in a hurry, you can use FERRET to send a message to the operator, to inquire whether the job has been run and whether there were errors.

An alternative procedure is to place these same "job" statements at the beginning of the "COURSE" file, or whatever file is the source file, before SYSTEM DIALOG; in this case the "M:SI" statement must be omitted. Then at the executive level (terminal prompts with '!') type

!ASSIGN M:SI,(FILE,COURSE)

The underlined characters are supplied by the computer. Enter BPM as just described to assemble your program, replying N to EDIT?. Another possibility, useful for long jobs, is to place the job 'cards' in a separate file, with the M:SI statement left in, and assign the job-card file as the source input file.

A successful assembly is necessary. METASYMBOL error messages identify sources of trouble, and the dialogue system also contains messages to assist the author. You should not be discouraged by the several assemblies needed for correcting errors. The row of asterisks at the lefthand margin on the printout indicates an error. Mostly errors are obvious on looking at them but occasionally the advice of an experienced programmer may be necessary. Very few programs of any complexity are initially without error, so a number of error runs are expected.

A common mistake is the use of a label more than once within the program. The assembler complains of a doubly defined symbol when you refer to such a label. You should give a new name to one of the offending statements and check the occurrences to see which label is needed when. A concordance, obtained during assembly, is useful because it shows where the offending label has been used.

place these same "job" statements
" file, or whatever file is the
OG; in this case the "M:SI"
at the executive level (terminal

URSE)
upplied by the computer. Enter
le your program, replying N to
eful for long jobs, is to place
ile, with the M:SI statement left
e as the source input file.

ary. METASYMBOL error messages
d the dialogue system also contains
You should not be discouraged by
or correcting errors. The row of
n on the printout indicates an
us on looking at them but occa-
ienced programmer may be necessary.
xity are initially without error,
xpected.

a label more than once within the
ns of a doubly defined symbol when
should give a new name to one of
eck the occurrences to see which
dance, obtained during assembly, is
he offending label has been used.

The 'find and type' command (FT) of EDIT is also very useful in tracking down labelling problems. Other common errors are the placing of a space after a comma, the omission of a quote, and the confusion of the letter O with zero.

A code is assigned to your program indicating the "severity" of the errors.

The binary file prepared by the assembler can be loaded using the LOAD subsystem at the terminal, and specifying the binary file (COURSEBO in the example above) as an "element file". The option U(B9999), where B9999 is the account with the dialogue library, is required. If on-line debugging using DELTA is desired by an experienced programmer option "D" is also needed. File assignments are made in the program, so only a carriage return is needed after "F:". Reply "Y(Carriage return)" after XEQ, and execution will begin. (If you use DELTA, ;G starts the program.)

4. Using Delta

Undoubtedly you will want to try the program, looking for bugs, after it has been successfully assembled. Keep the flowchart and the program listing available during this testing, making a point of checking at least the main branches. Testing of this kind will not discover all the bugs: only student usage will do that!

Delta is the name for a subsystem of BTM which can be a great help in testing your program, and is also the same for related facilities in the RUN and LOAD subsystems. It allows you to operate small

parts of the program, stopping to see what is in the counters and other storage locations. As with the other systems described here, Delta has more capabilities than we list. The facilities described here are enough to get a beginner started using the DELTA facilities in LOAD. Read the Delta chapter in the BTM manual to find other things which will be useful.

Let us assume that our program has been successfully assembled and is now on disk in binary form under the name COURSEBO. We have the program listing, the flowchart, and some notes on how we want to proceed with the testing. After signing on, we specify that we want to load a program. The dialogue with the machine will proceed as follows. (Underscoring indicates typing by the system.)

```
!LOAD

ELEMENT FILES:  COURSEBO
OPTIONS:  D,U(B9999)          (for delta)
F:                            (type carriage return only;
                              no additional files are
                              required)

SEV.LEV. = 0

**  NO UNDEFINED INTERNALS **
```

At this point a bell rings; Delta is ready for instructions. The primary facility available is the use of <u>breakpoints</u>. A breakpoint is a location in your program at which you wish the computer to stop, tell you where it is at and allow you to ask some questions. You will want to set breakpoints along all of the possible paths of the program segment you are interested in checking. Here are the commands to Delta controlling breakpoints:

e;B	(set the next available breakpoint at location e)
e,n;B	(set the nth breakpoint at e)

at is in the counters and
her systems described here,
. The facilities described
using the DELTA facilities
STM manual to find other

successfully assembled and
name COURSEBO. We have the
notes on how we want to
on, we specify that we want
e machine will proceed as
by the system.)

for delta)
type carriage return only;
o additional files are
equired)

dy for instructions. The
breakpoints. A breakpoint
ou wish the computer to
you to ask some questions.
ll of the possible paths of
n checking. Here are the
s:
le breakpoint at location e)
nt at e)

n;B (remove the nth breakpoint)

;B (type all of the breakpoints now in the program)

You might begin by typing a list like this:

ST1;B (the first breakpoint at label ST1)

M34+1;B (the next breakpoint one machine instruction past M37)

We assume the SD Metasymbol option here; it allows DELTA to recognize your labels. After running the program, you may wish to revise your strategy and remove or change these breakpoints. Then you would use the other forms listed above.

Having set up the breakpoints, it is necessary to start to run the program. The command

;G

causes it to start at its normal beginning. The command

ST1;G

causes it to start at label ST1. The program will proceed normally until it reaches one of the breakpoints specified. This will be announced with a line like this:

1;B ST1 (first breakpoint; at locations ST1)

It is now possible to examine the contents of various storage locations and machine registers, to be sure that things are going as expected.

e/ (display the contents of location e)

e(C/ (display e as a character string)

e(S/ (display e as a floating point number)

e(I/ (display e as an integer)

Line feed (display the word immediately following the one just displayed)

After the content of a word has been displayed, that word is considered to be "open." The user may type a new value for that word, hit the carriage return, and the new value will replace the one just displayed.

After the user is satisfied with the information he has received about the present breakpoint and the modifications he may have made, he may continue operating his program by typing

```
;P
```

or he may wish to begin again or start somewhere else using the ;G command, described above. When he is done working with his program and wants to return to the executive, he can accomplish this by two escapes.

5. <u>Generating a load module</u>

The version of the program to be used with students should be generated as a load module. Assuming that the programmer has successfully generated the binary file COURSEBO, without error and without the use of the SD option on the METASYM card, the following job will create the load module PROG1:

```
!JOB       (accounting information)
!LIMIT     (TIME,15)
!LOAD      (EF,(COURSEBO)),(LMN,PROG1),;
!(UNSAT,(B9999)),(BIAS,FA00),(ABS),(SL,9),;
!(PERM)
```

The JOB and LIMIT cards are the same in function as in previous examples. The options of the LOAD command require some definition.

l, that word is
new value for that
lue will replace the

ion he has received
tions he may have
y typing

ere else using the
working with his
he can accomplish

udents should be
e programmer has
BO, without error
ETASYM card, the
OG1:

G1),;
,(SL,9),;

tion as ir previous
equire some definition.

(Note that a semicolon is the run-on indicator.) Here are the LOAD arguments which are required:

> EF: the element files (in parentheses, separated by commas) which are to be put together to make up the load module. In our example, only one file. (Omitted if a GO file is used - see BTM manual.) Names of element files must have 8 or fewer characters.
>
> LMN: the load module name, eight or fewer characters.
>
> UNSAT: list of accounts (in parentheses, separated by commas) from which unsatisfied references are to be picked up. The library of each account is accessed. The account with the dialogue macros (B9999 at Irvine) must be included.
>
> BIAS: the lower limit into which on-line user programs can be loaded in this installation -- FA00 at Irvine; elsewhere, check with computer center personnel.
>
> ABS: specifies absolute load module.
>
> PERM: specifies that the file is to be permanently retained.

Here are some LOAD arguments which are <u>optional</u>:

> MAP: produces a listing of the locations into which the element files and external references are loaded. Very useful in debugging the program.
>
> SL: specifies the error severity level that will be tolerated by the loader in forming a load module. The value may range from 0 through F.

After the load module has been successfully generated, the programmer will want to run it. The procedure is:

> <u>!RUN</u> carriage return
>
> <u>LOAD MODULE FID:</u> PROG1 carriage return
>
> <u>;G</u> (and a bell, if terminal has one)

(Here again the machine printout is underlined.) The programmer now has a choice: to begin the program execution he may hit the

carriage return. If (as is often the case) the program still has errors, he may choose instead to use the DELTA facilities (explained in 3.4) to do some debugging or to set breakpoints for debugging at various points during execution. When he is ready to begin running the program, he types

;G carriage return

During the execution of the program, the user can always go into DELTA by pressing the escape key twice. When he is ready to proceed with the program he types

;P

to resume at the point he left (see 3.5 for variations). If he has set breakpoints the program will automatically stop at those points, ready for DELTA commands. He can proceed with the running of the program as above.

To stop the program before it finishes, the user can type STOP at any place where the program asks for an answer, or he can at any point hit 2 escapes twice in succession. This returns him to the Executive.

he case) the program still has
e the DELTA facilities (explained
set breakpoints for debugging
When he is ready to begin

, the user can always go into
ice. When he is ready to

3.5 for variations). If he
l automatically stop at those
He can proceed with the running

hes, the user can type STOP
for an answer, or he can at
ccession. This returns him to

6. <u>Sectioning a program</u>

Large programs present some special problems. They may take a very long time to assemble, or even refuse to assemble at all!! This is not only expensive, it may also be inconvenient: at some installations any long job will be held over and run at night. It is also easier to modify a program in many pieces. Thus the programmer may want to assemble his program in smaller pieces, reassembling them one at a time as errors are found. Some care must be taken to be sure that the relationship among the pieces or sections is made evident to the system. Each section must begin with

```
        SYSTEM     DIALOG
```

but only the <u>first</u> section (where the student begins the dialog) should have a NAME or START command. The last command to be <u>executed</u> in the program as a whole (not in each section) should be followed by a FINALE or STOP command. Each part other than the first section should end with

```
        END        (no arguments)
```

The first section should end with

```
        END     DIALOGUE
```

Any label which is referred to in one section and defined in another requires special treatment. If the two sections are called SOURCA and SOURCB, for instance, and the label A33 is defined in SOURCA and referred to in SOURCB, then the command

```
        DEF      A33
```

must appear in SOURCA; and the command

```
        SREF      A33
or      REF       A33
```

must appear in SOURCB. Any statement labels, or parameter names which are defined in one section and referred to in another require DEF statements (in the defining program) and SREF or REF statements (in the referring program). These statements can occur anywhere in the program sections, but it is good practice to put them at the beginning. All counters used <u>must</u> be defined in the first section, mentioned in a DEF statement in that section and in REF statements in other sections using the counter(s). Any SAVE COUNTER,ALL commands used in that section must be preceded by the COUNTER statement(s). More than one symbol can be included in each DEF or SREF statement. For example,

```
          SREF     A33,B1,B6,CC
```

takes care of the three labels A33,B1,B6, and the counter CC.

If ENTRY commands (for restart) are used in the second or other parts, ENTRY must also appear in the first section of the program, anywhere after the START or NAME command. In the other sections it can be used anywhere after the label to which the branch is made on entering that section. It is good practice to place it before a WRITE statement.

When each partial program is debugged, a load module to be used by students might be generated by this job:

```
          !JOB      (ACCOUNTING INFORMATION)
          !LIMIT    (TIME,10)
          !LOAD     (EF,(BINA),(BINB)),(LMN,LESSON1),(PERM),;
          !(BIAS,FA00),(ABS),(UNSAT,(B9999)),(SL,9),(MAP)
```

ement labels, or parameter names which
eferred to in another require DEF
ram) and SREF or REF statements (in
atements can occur anywhere in the
practice to put them at the beginn-
defined in the first section, mentioned
on and in REF statements in other
Any SAVE COUNTER,ALL commands used
by the COUNTER statement(s). More
In each DEF or SREF statement. For

B3,B1,B6, and the counter CC.

are used in the second or other
the first section of the program,
command. In the other sections it
bel to which the branch is made on
d practice to place it before a

gged, a load module to be used by
is job:
FORMATION)

NB)),(LMN,LESSON1),(PERM),;
AT,(B9999)),(SL,9),(MAP)

Note that the binary file BINA and BINB must be assembled without the SD METASYM option. If the total program size is large, !OLAY may be used instead of !LOAD, with the same arguments.

The following page shows an example of a pair of program sections with the control statements needed to get them assembled.

```
!JOB PHYSICS,IRVINE,2
!LIMIT (TIME,10)
!ASSIGN M:BO,(FILE,BINA)
!METASYM LS,LO,SI,BO,AC(B9999)
          SYSTEM      DIALOG
          NAME        'EXAM'
          DEF         CA,CM,AA,RDQ
          SREF        B1,XYZ,TAB1,TAB2
          COUNTER     (CA,CM,CTOT)
A1        etc.
          ...
          EPILOG
          END         DIALOGUE
-----------------------------------------------------------------
!JOB PHYSICS, IRVINE,2
!LIMIT (TIME,10)
!ASSIGN M:BO,(FILE,BINB)
!METASYM LS,LO,SI,BO,AC(B9999)
          SYSTEM      DIALOG
          DEF         B1,XYZ,TAB1,TAB2
          SREF        CA,CM,AA,RDQ
B1        etc.
          ...
          END
```

7. <u>Overlaying</u>

The SIGMA 7 BTM u
very well run int
of it essential!)
reorganize the pr
core, and two or
take turns occupy
'overlaying'. Th
related in time a

The root should h
since no part of
core. All counte
so the loader wi
access them by us

The 'root' also c

The command:

LOAD

will cause the se
into core, so tha
be followed by, c
to which the proc

LOAD

will cause BINA
tion (in BINA, u
The root program

7. <u>Overlaying</u>

The SIGMA 7 BTM user is allotted a limited amount of core, so he may very well run into the situation where he has written more coding (all of it essential!) than can be accomodated. It is often possible to reorganize the program into a 'root' segment, which is always in core, and two or more other segments, (or groups of segments), which take turns occupying the remaining available space. This is called 'overlaying'. The way in which the root and other segments are related in time and space is called the 'tree' structure.

The root should hold all information used by more than one segment, since no part of a segment is available for use when it is not in core. All counters should be defined in the root segment and DEFed so the loader will make them available to the others, which will access them by using an SREF statement.

The 'root' also contains the instructions for loading segments.

The command:

```
        LOAD      'BINA'
```

will cause the segment whose binary file is named BINA to be brought into core, so that the instructions in it can be executed. This must be followed by, or combined with, a branch to the location in BINA to which the programmer wants to transfer.

```
        LOAD      'BINA',BB1
```

will cause BINA to be loaded into core, and will also set the instruction (in BINA, usually) with label BB1 as the next one to be executed. The root program will have an

```
        SREF     BB1,
```

and segment BINA will have a

```
        DEF      BB1.
```

An alternate form of LOAD is TRANSFER.

The tree structure must be described in the !TREE control command immediately following !OLAY or !OVERLAY. The 'root' is the left-most segment in the command; from the root extend two or more 'paths', each consisting of those segments that may occupy core storage (along with the root) at the same time. Suppose we have our program assembled as 4 binary files, with BROOT the name of the root segment, B1 the segment that is to be executed first, B2A and B2B two segments that are to be loaded together into the same space B1 occupies, B1 another segment that is to be loaded into that space. Then !TREE command would be:

```
        !TREE     BROOT-(B1,B2A-B2B,B3)
```

The '-' indicates that the two named binaries can be loaded next to each other, at the same time, incore. the ',' indicates that two segments, (or groups of segments), are to overlay one another (that is, begin at the same core storage location when loaded). The '()' indicates a new level of overlay.

This tree statement says that at any given time we may have one of three different 'packages' in core storage:

1) BROOT and B1
2) BROOT,B2A and B2B
3) BROOT and B3

When a segment is not i
be referenced only by fi
always in core, it is us
are not in core simultan

The root segment would i

```
            SYSTEM    DIA
            NAME      'RT
            SREF      B1E
            DEF       R2,
            COUNTER   (CA
            WrITE     'TH
            WRITE     'LE
            LOAD      'B1

R2          WRITE     'LE
            LOAD      'B2

            LOAD      'B2
            etc.
            ...
            STOP
            END       DIA
```

The source file for B1 w

```
            SYSTEM    DIA
            SREF      R2,
            DEF       B1E
```

he !TREE control command
The 'root' is the left-
t extend two or more 'paths',
y occupy core storage (along
we have our program assembled
the root segment, B1 the
and B2B two segments that
pace B1 occupies, B1 another
ace. Then !TREE command

ries can be loaded next to
e ',' indicates that two
overlay one another (that
on when loaded). The '()'

en time we may have one of
e:

When a segment is not in core it is on disk, and anything in it can be referenced only by first loading it into core. Since BROOT is always in core, it is used for communication between sections which are not in core simultaneously.

The root segment would include:

```
           SYSTEM     DIALOG
           NAME       'RTEX'
           SREF       B1ENT,B2ENT,B3ENT
           DEF        R2,CADD,CMULT,CEXP,R4,R5
           COUNTER    (CADD,CMULT,CTOT,CEXP,CD)
           WrITE      'THIS IS A REVIEW OF COMPLEX NUMBERS.'
           WRITE      'LET''S TRY ADDITION FIRST.'
           LOAD       'B1',B1ENT                   (load file B1 and
                                                    start with the command
                                                    labelled B1ENT)
R2         WRITE      'LET''S TRY MULTIPLICATION'  (return from B1)
           LOAD       'B2A'                        (load B2A but do not
                                                    branch)
           LOAD       'B2B,B2ENT                   (load B2B and branch)
           etc.
           ...
           STOP
           END        DIALOGJE
```

The source file for B1 would include

```
           SYSTEM     DIALOG
           SREF       R2,CADD
           DEF        B1ENT
```

```
B1ENT     etc.
          ...
          TO      R2        (return to root)
          END
```

Similarly, each of the segments would contain SREFs for each of the labels and counters in the root to which it referred and DEFs for each of its symbols to which the root segment might refer. B2A and B2B must also, of course, contain DEF and SREF statements to define internal references between them.

The job cards for creating the load module COMP from these binary files would be:

```
!JOB (ACCOUNTING INFORMATION)
!LIMIT (TIME,10)
!OVERLAY (EF,(BROOT),(B1),(B2A),(B2B),(B3)),;
!(MAP),(PERM),(SL,9),(LMN,COMP),;
!(SEG),(UNSAT,(B9999)),(ABS),(BIAS,FA00)
!TREE ROOT-(B1,B2A-B2B,B3)
```

Here B9999 is the account in which the system library is stored, and FA00 is the lower limit of core storage for the program, which is a system parameter, and may differ in other installations. The command !OLAY could be used in place of !OVERLAY, with the same arguments.

8. Student Use

As indicated in 3.5, t
program by means of th
!RUN
LOAD MODULE
;G
The student must be to
the program and ;G.

When the student first
identification if the
is used for restarting
dialogue at a single si

When the student wants
usual procedure of pres
input, the word STOP.
allows restart, he is
next time to tries this

Because the use of the
non-programmers, at Irv
for calling dialogues.
types DI; then he types
No error messages or br
records of dialog usage
this may not be possibl
with considerable force

ain SREFs for each of the
t referred and DEFs for
ent might refer. B2A and
SREF statements to define

COMP from these binary

(B2B),(B3)),;

;

AS,FA00)

tem library is stored, and
r the program, which is a
installations. The command
with the same arguments.

8. Student Use

As indicated in 3.5, the students can use the conversational program by means of the RUN facility:

!RUN

LOAD MODULE FID: PROG1

;G

The student must be told how to sign on, to type RU, the name of the program and ;G.

When the student first enters the program, he is asked to type an identification if the restart facility is used. This identification is used for restarting purposes if the student does not complete the dialogue at a single sitting.

When the student wants to leave the terminal, he can follow the usual procedure of pressing Escape twice; or, he can type, at any input, the word STOP. If he enters STOP and if the program allows restart, he is reminded to use the same identification the next time to tries this dialogue.

Because the use of the RUN facility appears somewhat awkward to non-programmers, at Irvine we have installed a special subsystem for calling dialogues. At the prompt character (!) the student types DI; then he types the name of the dialogue he wishes to use. No error messages or break messages are sent to the student, and records of dialog usage are maintained. (At other installations, this may not be possible; system modifications are often resisted with considerable force.)

APPENDIX 1: EXAMPLES

1. Creating a binary file

```
!EDIT
*EDIT EXAMP
*TY 1-25                                         (type the symbolic file
   1.000 !JOB PCDP0006,ANNA,2                     previously entered)
   2.000 !LIMIT (TIME,5)
   3.000 !ASSIGN M:BO,(FILE,EXAMPBO)             (binary file to be called
   4.000 !METASYM SD,SI,BO,AC(B9999)              EXAMPBO)
   5.000          SYSTEM   DIALOG
   6.000 *        EXAMPLE OF A DIALOGUE PROGRAM
   7.000          NAME     'EXAM'
   8.000          COUNTER  COUNT
   9.000 A1       WRITE    'WHAT IS 4 X 5?'
  10.000          BUMP     COUNT
  11.000          INPUT
  12.000          IF       '20',A2
  13.000          IF       '9',A3
  14.000          OTHER    A7
  15.000 A3       TO       A5,(COUNT,3)
  16.000 A6       WRITE    'YOU''RE ADDING. TRY AGAIN.'
  17.000          TO       A1
  18.000 A7       TO       A5,(COUNT,GE,3)
  19.000          WRITE    'TRY AGAIN.'
  20.000          TO       A1
  21.000 A5       WRITE    '4 X 5 = 20'
  22.000          TO       A8
  23.000 A2       WRITE    'GOOD.'
  24.000 A8       EPILOG
  25.000          END      DIALOGUE
*END

!ASSIGN M:SI,(FILE,EXAMP)                        (assign the symbolic file to
                                                  system input)
!BPM                                             (call batch system)
INSERT JOB? Y                                    (user enters Y for yes)
YOUR MAXIMUM PRIORITY= 2
EDIT? N                                          (user enters N for no)
JOB INSERTED.  ID=9
STATUS CHECK? Y                                  (user enters Y for yes.  Status
ID=9                                              may be waiting, running, or
RUNNING.                                          completed)
ID=9
RUNNING.
ID=9
```

2. Loading and running

```
!LOAD
ELEMENT FILES: EXAMPBO
OPTIONS: P,U(B9999)
F:

SEV.LEV. = 0
*: NO UNDEFINED INTERNALS

;G

WHAT IS 4 X 5?
?9

YOU'RE ADDING. TRY AGAIN.
WHAT IS 4 X 5?
?20

GOOD.
YOU HAVE COMPLETED THIS PROG
PLEASE TYPE ANY COMMENTS AND
?NO COMMENT

THANK YOU
XIT AT #RSU1+.90

!
```

pe the symbolic file
viously entered)

nary file to be called
AMPBO)

AIN.'

ssign the symbolic file to
ystem input)
all batch system)
ser enters Y for yes)

ser enters N for no)

ser enters Y for yes. Status
ay be waiting, running, or
ompleted)

2. Loading and running a binary file

```
!LOAD
ELEMENT FILES: EXAMPBO                  (Load EXAMPBO, created above)
OPTIONS: D,U(B9999)                     (D for Delta; U(B9999) defines
F:                                       appropriate library)

SEV.LEV. = 0
*: NO UNDEFINED INTERNALS   *:

;G                                      (entered by user to start pro-
                                         gram operation)
WHAT IS 4 X 5?                          (beginning of the conversational
?9                                       dialogue)

YOU'RE ADDING. TRY AGAIN.
WHAT IS 4 X 5?
?20

GOOD.
YOU HAVE COMPLETED THIS PROGRAM.
PLEASE TYPE ANY COMMENTS AND SUGGESTIONS.
?NO COMMENT

THANK YOU
XIT AT ?RSU1+.90

!
```

3. Creating a Load Module

In this example, control information is typed directly and is not part of the EXAMP symbolic file, as it was in 1. Note that the SD option must not be used on the METASYM card in making the load module.

```
!EDIT
*EDIT EXAMP
*DE 1-4
```

Delete control commands from EXAMP file.

```
!RELEASE M:SI
```

Cancel any previous assignments.

```
!BPM
INSERT JOB? Y
YOUR MAXIMUM PRIORITY= 2
1
:!JOB PCDP0006,ANNA,2
2
:!LIMIT (TIME,5)
3
:!ASSIGN M:SI,(FILE,EXAMP)
4
:!ASSIGN M:BO,(FILE,EXAMPBO)
5
:!METASYM LS,SI,BO,AC(89999)
6
:!LOAD (LMN,EXAMPDI),(PERM),(BIAS,FA00),(ABS),(UNSAT,(89999)),;
7
:!(EF,(EXAMPBO))
```

(EXAMPDI will be name of load module)

```
8
:
EDIT? N
JOB INSERTED.  ID=35
STATUS CHECK? Y
ID=35
WAITING: 1 AHEAD
...
RUNNING.
ID=35
RUNNING.
ID=35
COMPLETED.
ID=

!GET RUN
```

Special command for Irvine system; usually !RUN is the command used to load EXAMPDI

```
LOAD MODULE FID:EXAMPDI
;G

WHAT IS 4 X 5?
?
```

(beginning of conversational dialog)

4. Using a FORTRAN S

```
!EDIT
*EDIT FORPLOT
*TY 1-10
   1.000 !JOB PHYSICS,IRV
   2.000 !LIMIT (TIME,5)
   3.000 !ASSIGN M:BO,(FI
   4.000 !FORTRAN SI,BO,L
   5.000        SUBROUTINE
   6.000        DIMENSION
   7.000        DO 1 I=1,2
   8.000 1      X(I)-SIN(I
   9.000        RETURN
  10.000        END
*
!ASSIGN M:SI,(FILE,FORPLO

!BPM
INSERT JOB? Y
YOUR MAXIMUM PRIORITY= 2
EDIT? N
JOB INSERTED.  ID=45
STATUS CHECK? N

!EDIT
*EDIT FORPLOTT
*SE1-15;/G0/S/20/;TY
--C2:NO SUCH STRG
   1.000 !JOB PHYSICS,IRV
   2.000 !LIMIT (TIME,10)
   3.000 !ASSIGN M:BO,(FI
   4.000 !METASYM SI,BO,L
   5.000          SYSTEM
   6.000          START
   7.000          REF
   8.000 AA       WRITE
   9.000 BB       FORTRAN
  10.000 CC       PLOT
  11.000          STOP
  12.000 X        RES
  13.000          END
--EOF HIT AFTER 13.
*
```

directly and is not
1. Note that the SD
n making the load

e control commands from
file.
l any previous assignments.

(89999));

PDI will be name of load
le)

ial command for Irvine system;
lly !RUN is the command used
oad EXAMPDI

inning of conversational
log)

4. Using a FORTRAN Subroutine

```
!EDIT
*EDIT FORPLOT
*TY 1-10
   1.000 !JOB PHYSICS,IRVINE,2
   2.000 !LIMIT (TIME,5)
   3.000 !ASSIGN M:BO,(FILE,VALUES)
   4.000 !FORTRAN SI,BO,LS,LO
   5.000        SUBROUTINE VALUES(X)
   6.000        DIMENSION X(100)
   7.000        DO 1 I=1,20
   8.000 1      X(I)=SIN(I/3.)
   9.000        RETURN
  10.000        END
*
!ASSIGN M:SI,(FILE,FORPLOT)

!BPM
INSERT JOB? Y
YOUR MAXIMUM PRIORITY= 2
EDIT? N
JOB INSERTED.  ID=45
STATUS CHECK? N

!EDIT
*EDIT FORPLOTT
*SE1-15;/60/S/20/;TY
—C2:NO SUCH STRG
   1.000 !JOB PHYSICS,IRVINE,2
   2.000 !LIMIT (TIME,10)
   3.000 !ASSIGN M:BO,(FILE,FORBO)
   4.000 !METASYM SI,BO,LO,AC(89999)
   5.000             SYSTEM    DIALOG
   6.000             START
   7.000             REF       VALUES
   8.000 AA          WRITE     'TEST OF PLOT'
   9.000 BB          FORTRAN   VALUES,X
  10.000 CC          PLOT      X,20
  11.000             STOP
  12.000 X           RES       100
  13.000             END       DIALOGUE
—EOF HIT AFTER 13.
*
```

File FORPLOT contains a FORTRAN subroutine, VALUES (x)

Compiling the subroutine

(FORPLOTT is a DIALOG program which calls VALUES. Modifying the program.

Line 9 will call the FORTRAN PROGRAM

Using FORTRAN Subroutines (Cont.)

```
!ASSIGN M:SI,(FILE,FORPLOTT)

!BPM
INSERT JOB? Y
YOUR MAXIMUM PRIORITY= 2
EDIT? N
JOB INSERTED.  ID=47
STATUS CHECK? Y
ID=47
WAITING: 8 AHEAD
CURRENT ID: 39
ID=47
WAITING: 8 AHEAD
CURRENT ID: 39
ID=47
WAITING: 8 AHEAD
CURRENT ID: 39
ID=47
COMPLETED.
ID=
```

Assemble FORTRAN routine.
Two alt modes to return to executive level.

Both binary outputs now available.

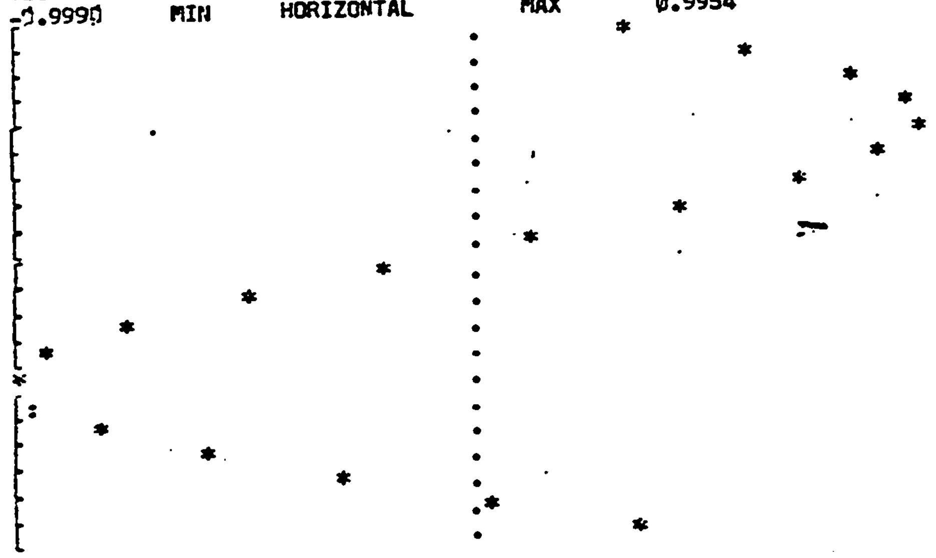
```
!LOAD
ELEMENT FILES: FORBD,VALUES
OPTIONS: U(B9999)
F:

SEV.LEV. = Ø
XEQ? Y
```

Run program on line.
Name both binary files.
Look for unsatisfied library references in account B9999.

Y says 'Proceed with execution'.

```
TEST OF PLOT
-Ø.999Ø     MIN     HORIZONTAL     MAX     Ø.9954
```



```
USER EXIT.
!
```

APPENDIX 2: REFERENCES

XDS SIGMA Symbol and Me

XDS Batch Timesharing M

XDS Batch Timesharing M

XDS Batch Processing Mo

ACKNOWLEDGEMENTS

The structure and devel
Estelle Warner, David R
programming contributio
Deering. The users who
software were Noah Sher
Ballard, and Charles Mu

APPENDIX 2: REFERENCES

XDS SIGMA Symbol and Metasymbol - 900952

XDS Batch Timesharing Monitor Reference Manual - 901577

XDS Batch Timesharing Monitor Users Guide - 901679

XDS Batch Processing Monitor Reference Manual - 900954

ACKNOWLEDGEMENTS

The structure and development of the software was guided by Estelle Warner, David Robson, and Alfred Bork. Substantial programming contributions were made by John Collins and Harold Deering. The users who heavily influenced the direction of system software were Noah Sherman, Mark Monroe, Kenneth Ford, Richard Ballard, and Charles Munch.

APPENDIX 3: A FINAL WORD (OR TWO) TO THE READER

* Comments on this manual, noting errors, omissions, and ambiguities will be appreciated.

* Copies of the system tape are available to those with SIGMA 7s who would like to try using it. Please enclose blank tape with your request.

* Those who are actively engaged in writing dialogs are asked to inform us of this fact so that we can keep them up-to-date on changes to the system as they occur. Such changes tend to be relatively minor and will be of small interest to any expect those actually using the system. Let us know the date of the latest modification you have.

* Dialogs which have been developed using this system are also available to potential users. Information will be sent on request.

* Reports of system errors or failures should be reported in detail, with copies of input and output, if possible.

* All such comments, requests, reports, and notifications should be addressed to:

Alfred M. Bork
Physics Computer Development Project
University of California, Irvine
Irvine, California 92664