$1.75

FOR SDS INTERNAL USE ONLY

TSD REFERENCE MANUAL

for

SDS SIGMA 7 COMPUTERS

90 15 20A

January 1968

SDS

SCIENTIFIC DATA SYSTEMS/1649 Seventeenth Street/Santa Monica, California

Printed in U.S.A.

CONTENTS

APPENDIXES

ILLUSTRATIONS

TABLES

EXAMPLES

1. INTRODUCTION

The TSD system is primarily a Time Shared Debugging aid. Using TSD, up to eight programmers can simultaneously perform checkout tasks, with each user acting as if he had exclusive use of the computing system.

TSD provides the following capabilities for such users:

1. Various RAD files can be dynamically assigned (and reassigned) for input/output purposes.

2. A source version of a program can be created or modified by use of convenient editing features.

3. Source programs can be assembled by use of an on-line version of SYMBOL during a debugging session.

4. An assembled program can be loaded and executed under TSD control.

5. Execution of assembled programs can be interrupted for examination and/or modification, and then continued or restarted at any point in the program.

6. Symbolic references can be used when examining or modifying a loaded program, and new symbols can be created and defined. In addition, the debugging language includes a limited subset of the SYMBOL language so that, while debugging, instructions can be coded in a symbolic form that resembles their source language counterparts.

HARDWARE

The TSD system has three components that are of immediate interest to the user:

1. A Sigma 7 with 32K memory (or more).

2. A RAD storage unit.

3. A user terminal.

The memory unit has 16K words set aside for the exclusive use of the on-line user. Programs occupying this "user core" run only in slave mode. Of the remaining memory, another 16K provides services and performs master mode control functions. The user cannot directly access this region, but certain system calls are available which obtain services from and modify tables in this region (see Appendix B).

If the Sigma 7 contains more than the minimum 32K memory, off-line (background) programs may be processed concurrently with on-line work. However, the user is independent of both background work and other on-line users. The RAD is the only device on which input/output files are maintained for on-line usage. Card reading, printing, card punching, and magnetic tape operations cannot be performed by the on-line user. However, RAD files can be set up off-line (see "FMGE" in the SDS Sigma 5/7 Batch Processing Monitor Reference Manual, Publication No. 90 09 54). These files can then be processed on-line or off-line and hard copies produced as needed.

The user terminal is a Teletype unit, but other terminals (e.g., keyboard/displays) could be used in future implementations of TSD. Presently, the system employs one model 35ASR Teletype (which has a paper tape punch and reader) and seven model 35KSR Teletypes.

The Teletypes provide the communication link between the user and the computer. Inputs are typed in by the user and outputs are printed on the Teletype. Figure 1 shows the keyboard layout of a 35KSR and Figure 2 shows the control panel. Appendix A contains a description of the controls and special keys.

TSD's time-sharing capability is accomplished by time-slicing and swapping. Eight on-line users (and possibly a background job) can share computing facilities during the same period of time. After a given burst of computing time, TSD stops servicing one user and gives service to another (or possibly to the background job).

The 16K user core is swapped via the RAD, i.e., the first user's software is written out and then a second user's software is read into user core. As a result of this procedure, a user will note that response occasionally is delayed. Ordinarily, the delay should be no longer than five seconds.

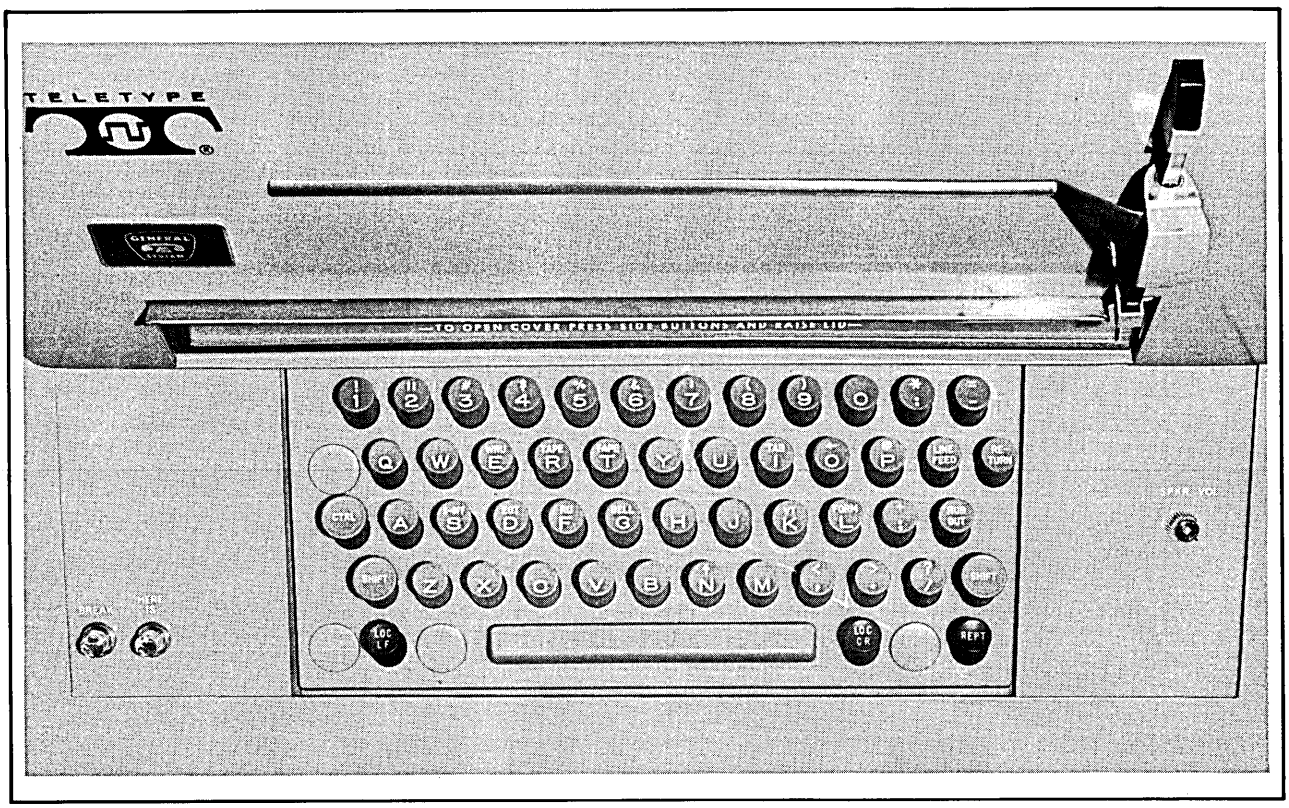

Figure 1. Teletype Keyboard

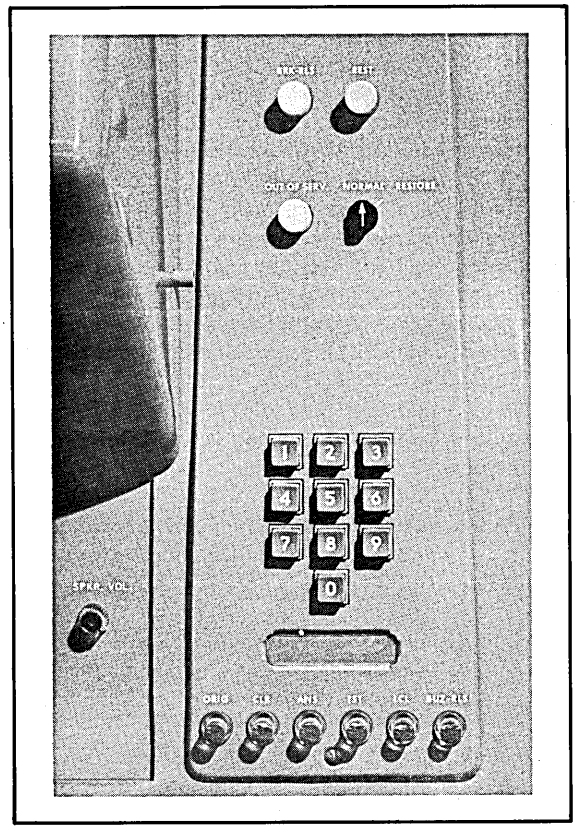

Figure 2. Teletype Control Panel

SOFTWARE

Five items of software are important to the TSD user:

1. TSD executive
2. EDIT subsystem
3. SYMBOL subsystem
4. DEBUG subsystem
5. User's program

The TSD executive is always available to the user. The executive can call (one at a time) any of the subsystems; i.e., EDIT, SYMBOL, or DEBUG. They, in turn, can automatically return to the executive. Only DEBUG can be used to load and execute the user's program. The user's program can then recall DEBUG. Figure 3 illustrates the hierarchy of the software.

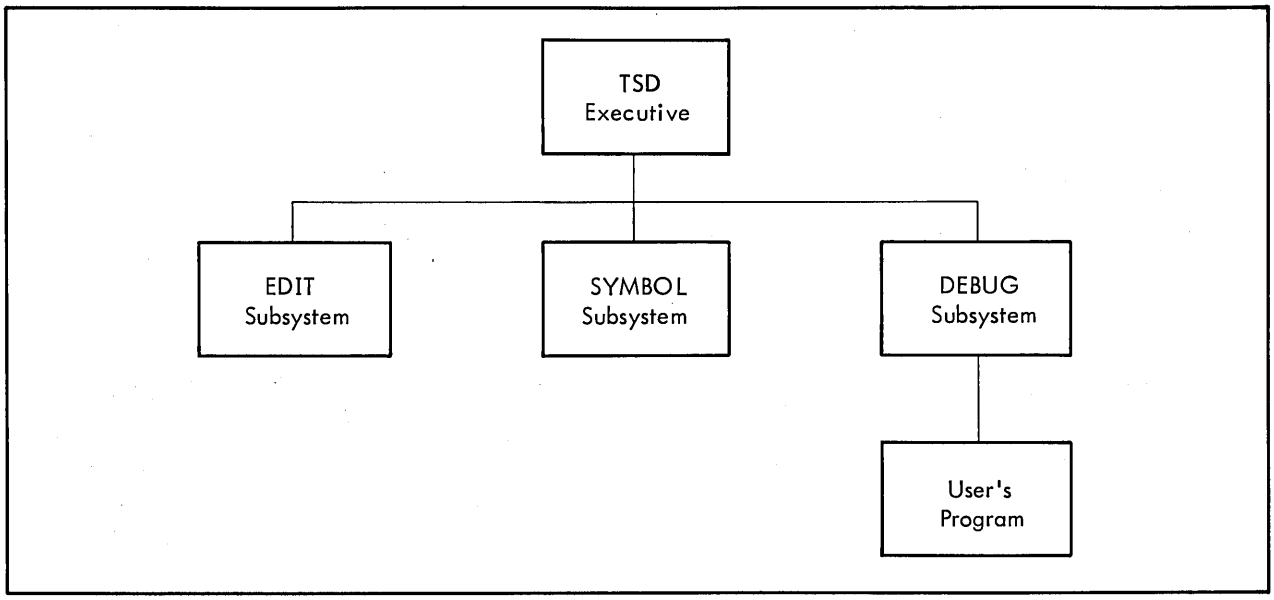

Figure 3. TSD Software Hierarchy

During service to the user, either the user's program or one of the three subsystems resides in user core. DEBUG and the user's program are never in core simultaneously. The TSD executive resides in the master portion of memory rather than user core, to ensure ultimate control for the user. The executive, therefore, is never swapped.

2. START-UP PROCEDURE

On-line service is obtained from TSD by turning on the user terminal and logging into the system.

TURN-ON PROCEDURE

The Teletype units are turned on by depressing the "ORIG" key (see Figure 2, Teletype Control Panel). The key locks in the depressed position and lights up. The key remains in this condition until the user Teletype is turned off (see "Signing Off", Section 3).

LOG-IN PROCEDURE

After the terminal is opened, the user alerts the TSD system by momentarily depressing the BREAK key (see Figure 1, Teletype Keyboard).

If the system is operative, one or more of the following messages will be typed:[†]

TSD SYSTEM IS UP – 10/15/67

The date shown identifies which version of TSD is in use. Following this message, the system types out a request for user identification.

!LOGIN:

The exclamation point (!) informs the user that he is communicating with the TSD executive and the colon (:) signifies a request for data, to which the user must respond. In this case, the data required is an 8-character (maximum) user identifier, followed by a period. TSD compares the user identifier with a table of valid identifiers. If the response is valid, further service may be obtained from TSD. If the identifier is invalid, TSD outputs a question mark (?) followed by another log-in request.

User identifiers may contain both letters and digits.

After a successful log-in, TSD outputs an exclamation point indicating that the system awaits commands to the TSD executive.

The following example shows a successful log-in procedure as viewed at the Teletype printer.

Example 1. TSD Log-In Procedure.

```
TSD SYSTEM IS UP – 10/15/67
!LOGIN: ABCD1234.
!
```

In this example, ABCD1234 is the user identification. Note that the period following the identification is required input from the user. Following the second exclamation point, the user may type any TSD executive command.

[†]Messages output by TSD are underlined throughout this manual for clarity. It should be understood, however, that actual Teletype output is not underlined.

3. TSD EXECUTIVE

The TSD executive provides control commands that influence the flow of the user's on-line work. There are six executive commands:

1. BYE
2. ASSIGN
3. EDIT
4. SYMBOL
5. DEBUG
6. PROCEED

BYE is used in signing off. ASSIGN lets the user specify RAD I/O files to be used in on-line processing. EDIT, SYMBOL, and DEBUG constitute subsystem calls, and PROCEED is used to continue an interrupted subsystem process.

There is a high degree of interplay between the TSD Executive and the user during executive command input.

First, the Executive must be in control; this is signified by the appearance of an exclamation point at the teleprinter.

Second, the user types two letters that begin an Executive command, namely: BY, AS, ED, SY, DE, or PR.

Third, TSD types the remaining letters of the command (in cases of confusion, the executive outputs a question mark and starts a new command line with another exclamation point).

Fourth, the user supplies parameters, if any, and confirms the command by typing a period. The user can scratch the command by depressing the BREAK key any time prior to typing the confirming period. Scratched commands are never executed, and the executive responds by typing an exclamation point to indicate readiness for a new command.

Discussions of each executive command are given below.

BYE Signing Off

An on-line work session is terminated by giving the executive command BYE and by turning off the user terminal. The correct form of the BYE command shown below assumes that the TSD executive is in control. Note the exclamation point.

!BYE.

The user turns off the terminal after completing the BYE command by momentarily depressing the "CLR" key (see Figure 2, Teletype Control Panel).

ASSIGN File Assignment

The ASSIGN command is used to control all RAD file assignments for TSD users. This command has format and options similar to the ASSIGN control card in the Batch Monitor (see SDS Sigma 5/7 Batch Processing Monitor Reference Manual, Publication No. 90 09 54). A general form of the command is shown below. Items in capital letters are required as shown; items in lower case letters represent parameters. Brackets denote optional items.

!ASSIGN dcb-name, (FILE, file-name[, acct. no.])[, (option)...].

where

dcb-name is a name of up to eight characters formed from the SYMBOL character set and having as its first two characters either F: or M:. The F: identifies files utilized in the user's program. The M: identifies system files. Names used at present in TSD operations are

M:SI	Source input	M:BO	Binary output
M:SO	Source output	M:LO	Listing output
M:BI	Binary input	M:C	Command file (for EDIT)

file-name is a user-selected identifier of up to eight letters or digits.

[,acct. no.] is an arbitrary identifier that designates the named file as a user-account file. There are two kinds of files; TSD temporary files and user-account files. If no account number is given, a TSD temporary file is used. During on-line operation, these files are reserved for the user, but the user cannot save them for off-line processing. Consequently, TSD temporary files "belong" to a user only until the user signs off.

User-account files are specified by giving any acct. no. of up to eight characters comprised of letters or digits.

User-account files can be saved for off-line processing (e.g., printing or punching) or saved for future on-line sessions. (The user is referred to the Batch Monitor Manual, cited above, for complete descriptions of Batch file utilities – see FMGE.)

[,(option)]. represents the various ASSIGN options recognized by the Monitor. Those allowed to the TSD user are given in Table 1 below.

Table 1. Allowable ASSIGN Options for TSD

Option	Purpose
(IN) (INOUT) (OUT) (OUTIN)	These options describe the function of an assigned file. The meanings are identical to their Batch Monitor counterparts.
(REL) (SAVE)	Same meaning as in Batch, i.e., release or save the file.
(EXPIRE, NEVER)	No expiration date is allowed. If this option is not given, the default Batch value is used.
(PASS, value)	Defines a password. The "value" may be up to eight characters.
(READ, ALL)	All accounts have read access to the given file.
(WRITE, ALL)	All accounts have write access to the given file.
(READ, NONE)	No other account number may read the given file.
(WRITE, NONE)	No other account number may write in the given file.

Files created off-line for TSD work must specify that either ALL accounts have access or, at least, that the TSD account number (:TSD) is allowed access. Otherwise, TSD (which operates as a foreground program) cannot access files even though user account numbers agree. It is recommended that unlimited access (ALL) be specified for files involving TSD work, but that passwords be used if privacy is a consideration.

A sample ASSIGN command is shown below:

```
!ASSIGN M:SI, (FILE, INPUT, ACCT29), (PASS, FRIEND).
```

The above command states that the file named "INPUT" was created off-line under account number "ACCT29". It is to be used as a source input file. If the created file does not permit the password "FRIEND", TSD is not allowed to access the file, and the user will receive an error message. Other examples of ASSIGN usages are given in subsequent sections.

EDIT, SYMBOL, DEBUG Subsystem Calls

Proper file assignments must be given before the EDIT, SYMBOL, or DEBUG subsystems are called. In Sections 4, 5, and 6, the assignments appropriate for each subsystem are detailed. In the following sample subsystem calls, it

is assumed that the TSD executive is in control, and that proper assignments have been made. The correct forms for the three subsystem calls are

!EDIT.

!SYMBOL.

!DEBUG.

Once a subsystem call is made, the executive is no longer in control and control has been transferred to the given subsystem. In some cases the subsystems automatically return control to the executive, but generally, the user must initiate an action that results in control returning to the executive. This topic is fully covered in Sections 4, 5, and 6; however, the remainder of this section briefly explains how a user can return control to the executive.

BREAK AND PROCEED

The BREAK key returns control to the TSD Executive. If a subsystem has control, a single BREAK causes TSD to put the Executive in control. If the user's program is being executed, one BREAK returns control to the DEBUG subsystem (see Section 6), and a second BREAK puts the Executive in control.

Once the Executive has regained control, any Executive command may be given. Two commands are of special interest to the user when subsystem operation has been interrupted. If the user wishes to make a fresh start in a subsystem that has just lost control, a new call is issued to the subsystem (possibly after changing file assignments if the user desires). If the user wishes to continue subsystem operations rather than restart, the PROCEED command is used.

PROCEED Continue Subsystem Operation

The correct form of the PROCEED command is

!PROCEED.

This command is used only to continue an interrupted subsystem process. Two steps must be taken to continue execution of the user's program. First, the user must put DEBUG in control with a PROCEED command. Second, the user must issue the DEBUG proceed command (see Section 6). Note that once a new subsystem is called, an old subsystem cannot be continued. It can only be restarted.

4. EDIT SUBSYSTEM

The EDIT subsystem permits the user to start with a source (EBCDIC) input file and to produce an updated source output file. The source files are treated as a sequence of card images in which card columns 74 through 80 contain sequence numbers. Under user control, EDIT copies cards from the input file to the output file until a sequence number is encountered at which the user has indicated a deletion, replacement, or insertion. After making the desired change, EDIT continues copying until another specified sequence number is reached. The user must put the desired changes in the proper order before exercising EDIT. Once a given card image has been output, EDIT will not back up to permit changing earlier images. If it becomes necessary to make such changes, a new EDIT call may be used, but this is an inefficient and time-consuming procedure.

It is important to note that EDIT does not modify the input file. Thus, if a user finds that his updated output file is incorrect, he may start again, with the original input file intact.

Sequence numbers are 7-digit decimal constants. EDIT places a decimal point between the fourth and fifth digit; thus, if card columns 74-80 contain 1234567, EDIT considers the sequence number to be 1234.567. Therefore, EDIT deals only with the following range of sequence numbers – 0000.000 through 9999.999. For the user's convenience, EDIT automatically generates sequence numbers when inserts are made on-line.

RESEQUENCE Output New Sequence Numbers

The RESEQUENCE command enables the user to generate correct sequence numbers in a file that, though correctly ordered, contains incorrect sequence numbers, or is blank in columns 74-80.

Two parameters are required for a RESEQUENCE command: a start number and an increment. The user may or may not wish to provide these parameters; implicit, or default, values are used by EDIT if the user omits one or both parameters. The implicit increment is 1.000, and the implicit start value is usually 1.000 (refer to the later discussion "Completing An Update" for an exception).

There are many acceptable forms for RESEQUENCE (this is true of all EDIT commands). An individual user may adopt a form that fits his requirements. The permissible variations are as follows:

1. Parameters may be placed either before or after the command designator. (Command designators are the characters that uniquely identify each command. The command designator for RESEQUENCE is R.)
2. Parameters may be separated by a comma, a slash, a colon, or any other punctuation mark except a period.
3. Command designators may be followed by any number of other letters at the user's discretion.

The RESEQUENCE examples below illustrate various forms that may be used. The ">" sign preceding each example is output by EDIT, and therefore, is underlined in this manual. It is used to inform the user that EDIT has control and awaits a command.

Cr (Carriage Return)

The "Cr" following each example for the EDIT subsystem represents the Carriage Return. This key signifies the end of an EDIT command.

Example 2. RESEQUENCE Commands.

```
> 10/.1RESEQUENCE Cr
> 0/2RESEQ Cr
> 100,1R Cr
> 100R Cr
>,1R Cr
>RESEQ10/.1 Cr
>R100:2 Cr
>R100 Cr
>R,2 Cr
>RESEQUENCE Cr
>R Cr
```

The user may infer similar forms for other EDIT commands.

The following printout shows a typical example of RESEQUENCE as used by a new on-line user.

Example 3. Use of RESEQUENCE.

```
TSD SYSTEM IS UP – 10/15/67

!LOG IN: AJC24

!ASSIGN M:SI, (FILE, OLDE, ACCT29).
!ASSIGN M:SO, (FILE, NEW, ACCT29).
!EDIT.

>RESEQ 10/1Cr

!
```

In the above example, the ASSIGN commands indicate that the two files named OLDE and NEW were created off-line under the account number ACCT29. The input file, OLDE, contains card images that are copied onto the NEW file after new sequence numbers are generated. The EDIT subsystem generates the sequenced images onto the NEW file, overwriting any previous information there. The RESEQUENCE command specifies that the first NEW card image has sequence number 0010000, the second has 0011000, the third has 0012000, etc. After the last card image is placed in the NEW file, EDIT concludes resequencing by returning control to the TSD Executive. Therefore, after any resequence is finished, an exclamation point is printed.

TYPING ERROR RECOVERY DURING EDIT

The EDIT subsystem provides two control mechanisms for correcting typing errors: the "erase" and "backspace" control functions.

W^C Erase.

To erase the current line, the user employs the control-W function (designated as W^C). To perform any control function, the user depresses and holds the CTRL key followed by the appropriate control designator (in this case W). Since W^C erases only the current line, it must be exercised before the carriage return ending that line. Example 4 shows a printout when W^C is used.

Example 4. W^C Correction.

```
> 10/2RESEQU@@
>
```

In the example, the user depressed W^C after typing "U"; EDIT responded by outputting two @ signs, "erasing" the current line, and starting a new line.

H^C Effective Backspace.

The other EDIT control function for correcting typing errors is the effective "backspace". To backspace, the user types a control-H (designated H^C) in the same manner described for W^C. Example 5 illustrates one use of H^C.

Example 5. H^C Correction.

```
> RESEQ10/1 # 2 Cr
```

In the example, the user entered a single H^C to replace the increment "1" with "2". After typing the "1", the user immediately used the H^C; EDIT responded by typing a # sign and effectively backspaced over the unwanted "1". More than one character may be spaced over by using successive H^C type-ins. Example 6 illustrates this situation.

Example 6. Successive H^C Commands.

```
> RESEQ1/10####10/2 Cr
```

In the example, the user entered four successive H^C characters to replace the entire parameter list. The resulting line becomes "RESEQ10/2 Cr". Note that H^C only applies to the current line being typed. The user cannot "backspace" over a carriage return.

UPDATING SOURCE IMAGES

There are three EDIT commands used in updating: DELETE, COPY, and APPEND. Designators for these commands are: D, C, and A, respectively. In the following explanations of these commands, examples are given to clarify their use. In all the examples, assume that the user has entered EDIT as follows:

```
!ASSIGN M:SI,(FILE,INP,ACCT29).
!ASSIGN M:SO,(FILE,UP,ACCT29).
!EDIT.

>
```

Also assume that the input file, INP, has the following contents:

Columns	
1 73	74 80
A10	0010000
A20	0020000
A30	0030000
A40	0040000
A50	0050000
A60	0060000
A70	0070000
A80	0080000
A90	0090000

DELETE Delete Source Images

The DELETE command has two parameters that specify the starting and ending point of its operation. Example 7 illustrates use of the DELETE command (refer to the sample input file above).

Example 7. DELETE Commands.

```
> DELETE,20 Cr
> DELETE50,60 Cr
> DELETE,90 Cr
```

Note the comma following the DELETE in the first and last commands, which implies a "blank" first parameter. This defines the starting location for the command as "current point".

The first command skips over all cards in the INP file until the card having sequence number 0020000 is skipped.

The second command copies cards from the current point in the INP file until sequence number 0050000 is found; all cards from 0050000 through 0060000 are then skipped.

The third command causes a skip over all cards from the current point (0060000) until 0090000 is skipped. As a result of these three commands, the output file UP contains only the two cards numbered 0030000 and 0040000.

Note that the following forms of the DELETE command are equivalent:

```
> DELETE30 Cr
> DELETE30, Cr
> DELETE30,30 Cr
```

In each case, only card 30 is deleted.

If this command were given at the beginning of an update process, all cards up to 0030000 would be copied onto the output file, card 0030000 would be deleted, and the EDIT subsystem would then wait for further commands.

COPY Copy Source Image

The COPY command, which is the complement of the DELETE command, also has two parameters that define the starting and ending point of its operation. In the example below, assume that the user starts with a new update; i.e., the break key has been pushed, the TSD Executive has gained control and typed an exclamation point, and the user has entered a new EDIT command. This effectively rewinds the assigned output file UP and the input file INP.

Example 8. COPY Commands.

(BREAK key is used.)
!EDIT.

> COPY,20 Cr
> COPY50,60 Cr
> COPY,90 Cr

The first command copies all cards in input file INP until the card having sequence number 0020000 is placed in the UP file.

The second command skips all INP cards from the current point until sequence number 00500000 is found; then cards from 0050000 through 0060000 are copied to the UP file.

The third COPY command causes all cards from the current point through 0090000 to be copied. In this particular example, the only INP cards not copied to the UP file are those with sequence numbers 0030000 and 0040000.

Note that with both DELETE and COPY a single parameter must be preceded by a separator (e.g., a comma).

APPEND Append Source Image

The APPEND command is used for inserting new cards. It may have one, two or no parameters. The first parameter states the sequence number at which inserts are to be placed. All cards from the last one copied (if any) up to the sequence number given are automatically copied to the output file. If this number is not given, inserts are placed immediately after the last card written out.

The second parameter is an increment used to establish new sequence numbers (that are automatically generated for the user). If the increment is not given, EDIT uses 1.000.

The APPEND command may be used to replace cards, but if successive cards are to be replaced, it is recommended that a DELETE precede the APPEND.

In replacing cards, APPEND deletes input cards falling within the range of the new cards being input. However, if a short routine is intended to replace a larger one, "left-over" cards may be inadvertently copied if a DELETE does not precede the APPEND.

Once an APPEND command is given, EDIT starts a new line, generates a sequence number, and awaits the text to be inserted. Each new line of text (i.e., card columns 1-73) ends when the user types a carriage return. EDIT then starts another line, generates the next sequence number, and awaits the next line of text.

C^C End Text Insert

An APPEND process may be continued until the user signals completion of the current text insert by typing control-C (designated C^C).

Example 9 illustrates the important features of APPEND. Assume again that the user starts a new EDIT.

Example 9. APPEND Commands.

(BREAK key is used.)
!EDIT.

> APPEND Cr
1.000 B1 Cr
2.000 B2 C^c
> APPEND 15 Cr
15.000 B15 Cr
16.000 B16 C^c
> DELETE20,40 Cr
> APPEND,3 Cr
19.000 B19 Cr
22.000 B22 Cr
25.000 C^c
> APPEND79,3 Cr
79.000 B79 Cr
82.000 B82 C^c
>

The contents of the output file that results from the preceding sequence of commands is shown below. Refer to the sample input file shown previously for comparison.

Columns	
1 ... 73	74 ... 80
B1	0001000
B2	0002000
A10	0010000
B15	0015000
B16	0016000
B19	0019000
B22	0022000
A50	0050000
A60	0060000
A70	0070000
B79	0079000
B82	0082000

It should be noted that the above output file is not necessarily complete; the user is free to make further additions or to continue copying (e.g., A90 in the sample input file).

COMPLETING AN UPDATE

There are three commands used to complete the update: STOP, END, and RESEQUENCE.

STOP Stop Update

The STOP command (designator S, no parameters) is used if no further cards are to be copied.

END End Update

The END command (designator E, no parameters) is used if the remainder of the input file is to be copied at the end of the output file. After STOP or END command processing, EDIT closes the input and output files and returns control to the TSD Executive. Thus, final completion of an update is confirmed by the exclamation point produced by the Executive.

The RESEQUENCE command offers another method for the user to complete an update. When this command is entered, EDIT copies the remaining input cards after assigning sequence numbers as dictated by the RESEQUENCE parameters. If no start number is specified (parameter number 1), the RESEQUENCE increment is added to the latest sequence number placed in the output file. From that point on, RESEQUENCE continues as previously described.

Note that the control functions H^C (Backspace) and W^C (Erase) may be used to modify not only command inputs but text inputs as well.

Example 10. H^C and W^C Text Corrections.

```
> APPEND35,.5 Cr
35.000  C3355###5 Cr
35.500  C35.5@@
D35.5 C^C
>
```

In this example, the user gave three successive H^C inputs to correct line 35.000, and used W^C to scratch the first attempt at line 35.500. The result is equivalent to the following:

```
> APPEND35,.5 Cr
35.000  C35 Cr
35.500  D35.5 C^C
>
```

ORIGINATING SOURCE INPUT ON-LINE

ORIG Originate Source File

Although the primary function of EDIT is to facilitate source updates, EDIT also provides a command for originating a source program at the terminal. The ORIGINATE command (command designator O) is essentially the same as the APPEND command. However, when ORIGINATE is used, there is no RAD input file since the source of input is the terminal. Example 11 illustrates on-line origination of the same input file used in previous examples. Assume that the TSD executive has initial control.

Example 11. On-line Origination.

```
!ASSIGN M:SO,(FILE,INP,ACCT29).
!EDIT.

> ORIG10,10 Cr
10.000  A10 Cr
20.000  A20 Cr
30.000  A30 Cr
40.000  A40 Cr
50.000  A50 Cr
60.000  A60 Cr
70.000  A70 Cr
80.000  A80 Cr
90.000  A90 C^C
!
```

In example 11, note that the ORIGINATE is terminated by C^C (end text insert). This is the only way to complete an origination; once C^C is given, EDIT closes the file and returns control to the TSD Executive.

Lf (Line Feed)

The Teletype limits printing to 72 characters per line. This sometimes proves restrictive in using the ORIGINATE and APPEND commands because a print line will not hold the necessary number of characters (the Teletype simply overprints in its last column). A special character is available in EDIT to circumvent this difficulty. The keyboard character LINE FEED (designated Lf) causes EDIT to return to a new line. Thus, subsequent characters are visible as the user inputs them, and the new line is considered by EDIT to be a continuation of the current card. If more than 73 characters are input, EDIT ignores those beyond column 73 (of the input image).

USING THE BREAK KEY DURING AN EDIT

As stated previously, the BREAK key can be depressed to interrupt current processing. During an EDIT, BREAK causes control to be returned to the TSD Executive and this is the quickest method to abandon an update and make a new start (see the COPY example above). Then, giving the Executive command EDIT effectively causes the user's input and output files to be rewound.

Another reason for using BREAK during an EDIT is that occasionally the user will start an update only to realize that something has been overlooked.

Recovery is sometimes possible in this situation because of the relative slowness of the I/O operations involved. Three steps must be taken:

1. The EDIT in progress is stopped via BREAK. This places the Executive in control.

2. Control is then restored to EDIT via the PROCEED command. When PROCEED is used, EDIT is placed in control but not reinitialized. Furthermore, the update in progress at the BREAK is not automatically resumed.

3. The user must determine how far the update progressed. A special EDIT command, INQUIRE, achieves this.

INQUIRE Inquire Update Status

The INQUIRE command (command designator I, no parameters) prints for the user the sequence number and contents, respectively, of the last card processed by EDIT. That card image is no longer accessible (on the current EDIT), but its successors may be copied, deleted, or replaced. Example 12 illustrates this procedure. (Refer to the previous sample input file.)

Example 12. INQUIRE Command.

```
!EDIT.
> DELETE 30,90 Cr
(BREAK is activated immediately)
!PROCEED.

> INQUIRE Cr
50.000
A50
> DELETE,80 Cr
> END Cr
```

In Example 12, the user was able to stop the deletion process after card number 50. The user then had to issue a second DELETE command to remove the remaining unwanted cards, A60, A70, and A80.

There is no timing guarantee given for a BREAK. If data is being printed when the BREAK occurs, printing may continue for a short time. In no event should BREAKs be given in rapid succession because loss of RAD output file data can occur. One BREAK is usually sufficient.

ADDITIONAL EDIT FEATURES

In addition to the primary features already described, EDIT has a number of secondary features, including a set of mode commands: LIST, NOS., and TABS. Mode commands differ from other EDIT commands in that they set a mode of operation rather than cause an action. Mode commands depend on other commands for their effect to become evident.

LIST List Output Images

When the LIST mode is initially activated, nothing is printed. However, as other EDIT commands occur, LIST causes a printout to be produced every time a card image is moved from the source input file to the source output file. Example 13 shows the effect of LIST during an EDIT session. It is assumed that the input file is the same as that used on previous EDIT examples.

Example 13. LIST Command.

```
!EDIT.

> LIST Cr
> DELETE30, 80 Cr
10.000A10
20.000A20
> END Cr
90.000A90
!
```

The LIST mode is initiated by typing the command designator L and, optionally, preceding it with a plus sign, e.g., +L. To turn off the mode, the user precedes the command designator with a minus sign, e.g., -L. The LIST mode is initially off.

NOS. List Sequence Numbers

The NOS. command is used to decrease the EDIT printout. It will be recalled that when APPEND or ORIGINATE are used, sequence numbers are automatically generated. If the NOS. mode is on, these sequence numbers are printed out at the terminal. To eliminate this printout, the user precedes the command designator with a minus sign, e.g., -N. The NOS. mode can be turned on again by typing +N or simply N. Example 14 illustrates both the on and off mode for NOS.

Example 14. NOS. Command.

```
!EDIT.

> -NOS. Cr
> APPEND15 Cr
F15C^C
> +NOS. Cr
> APPEND25 Cr
25.000 F25 Cr
26.000 F26 C^C
>
```

In example 14, it should be emphasized that the sequence number for the card F15 is, in fact, 0015000 although nothing prints to indicate it. The user must exercise great care in further updating if he has turned off the NOS. mode. The NOS. mode is initially on.

TABS Tab Set or Clear

The TABS command differs from LIST and NOS. commands in that the TABS mode is always on; the user determines whether to use it or not. The TABS command permits the user to assign tab stops at various columns of a card image. Initially, the tab settings are 10, 19, and 37 which correspond to SYMBOL keypunch columns. The user sets tab stops by entering the TABS command followed by numbers designating the columns at which stops are to be set.

I^C Tab

The user exercises a tab by activating control-I (designated I^C). The upper part of the I key is labeled "TAB" (see Figure 1, Teletype Keyboard).

When I^C is used, EDIT spaces the printout over to the next tab column. If there is no next tab column, EDIT does not space but rings the terminal's bell instead. Tabs are cleared by giving the command (designator T) followed by a carriage return. Example 15 illustrates the setting and clearing of tabs.

Example 15. TABS Command and I^C Tabulate.

```
!EDIT.
> TABS6, 10 Cr
> APPEND Cr
1.000 A_ _ _ _BC_ _D C^C
> TABS Cr
>
```

In the above example, the user input an "A" followed by I^C. EDIT then spaced over to column 6. After typing "BC", the user again typed I^C, and EDIT spaced over to colmn 10. The last TABS command cleared the tab stops.

Of the six control characters provided by EDIT, W^C (Erase), H^C (Backspace), C^C (End Insert) and I^C (Tab) were explained previously. The remaining two EDIT controls are B^C (Retype) and V^C (Literal Next).

B^C Retype

B^C causes a line to be retyped. This feature is especially beneficial to the user who has been making H^C (Backspace) corrections and who wishes to view the corrected version of the line. For example,

```
> TES###RESEQ100#,.5<<
```

prints as follows if B^C is used before a carriage return is given:

```
> RESEQ10,.5
```

After a line is retyped, the user may further correct or add to it. A carriage return would probably be added in the above example. Note, in the example, that EDIT outputs "<<" to signify that B^C has been used.

V^C Literal Next

The V^C control can be used to place EDIT controls into the user's output file. When V^C is given, the subsequent character is not interpreted as an EDIT control but is stored as a noncontrol character. In other words, a V^C causes the next character to be accepted literally.

FILE Take Updates from Command File

The FILE command (designator F, no parameters) is used for large volume updates. To invoke this capability, the user types an F or + F. EDIT then takes its commands from a command file (see "CMDFILE" below) rather than from the terminal. To use this capability, an EBCDIC command file, which contains EDIT commands and text, is prepared off-line. This file has almost the same apperance as if the user performed the update on-line (exceptions are discussed below).

The advantage of the FILE command is that the user may prepare updates off-line and monitor the results on-line without laborious typing at the terminal.

Example 16 shows how this command may be invoked at the terminal.

Example 16. FILE Command.

```
!ASSIGN M:C, (FILE, CMDFILE, ACCT29).
!ASSIGN M:SI, (FILE, OLD, ACCT29).
!ASSIGN M:SO, (FILE, NEW, ACCT29).
!EDIT.

> FILE Cr
.
.
.
```

To terminate the FILE command, the command file should end with the command designator preceded by a minus sign, i.e., -F. However, the user may also interrupt this activity by depressing the BREAK key; the file command is then disabled. If the user then issues the PROCEED command, EDIT will accept further input from the terminal.

When keypunching a command file, two exceptions must concern the programmer: carriage returns and the C^c control used in terminating text input under the APPEND command. The user is free to omit carriage returns on command file cards; however, if he wishes to give them explicity, an 11-5-9 punch is used. The character C^c is generated by a 0-6-9 punch. Table 2 lists required punching for other EDIT controls and special characters.

Table 2. EDIT Controls on Punched Cards

Character	Function	Hexadecimal Equivalent	Card Punch
Cr	Carriage Return	X'15'	11-5-9
Lf	Line Feed	X'25'	0-5-9
B^c	Retype	X'14'	11-4-9
C^c	End of Text	X'26'	0-6-9
H^c	Effective Backspace	X'16'	11-6-9
I^c	Tab	X'05'	12-5-9
V^c	Literal	X'27'	0-7-9
W^c	Erase	X'07'	12-7-9

5. SYMBOL SUBSYSTEM

The TSD Symbol subsystem is a slightly extended (on-line) version of SDS Sigma 5/7 Symbol. The standard Symbol system assembles source input (SI) to produce two files: binary output (BO) and listing output (LO).

TSD's (on-line) Symbol can be directed to produce the same results, but is usually used to provide a load file for on-line debugging. Therefore, the user will want a BO file that contains a symbol table for debugging purposes, in addition to the usual BO data.

SYMBOL OPTIONS

Other options are available to the TSD Symbol user that can be used or declined as desired. They are:

Request or decline a BO file.

Request or decline an LO file.

Request or decline a listing at the terminal (TSD Symbol prints error messages at the terminal in any case).

There is a close relationship between the options selected and the file assignments the user must make before calling the Symbol subsystem. This is reflected in the examples shown below.

Once Symbol subsystem is called, it responds immediately with a request for options. The user then types an option list in which the options are separated by commas, and the list terminates with a period. Any time prior to typing the period, the user may give a carriage return to erase the option list. Symbol then prints a new option request. The following table gives the form of each option. Options underlined in the table are default options. If the user desires a default option, it is not necessary to supply that option in the option list.

Table 3. Symbol Subsystem Options

Description	Option Form	
	Wanted	Not Wanted
BO produced	BO	NOBO
LO produced	LO	NOLO
Debugging Symbol table on BO	DB	NODB
Terminal listing	TL	NOTL

Example 17. Binary Output with Debugging Symbol Table, No Listings.

```
!ASSIGN M:SI,(FILE,SOURCE,ACCT29).
!ASSIGN M:BO,(FILE,OBJECT).
!SYMBOL.
OPTIONS .
:
:
```

Example 18. Only Diagnostic Messages Desired, No Listing or Binary Output.

```
!ASSIGN M:SI,(FILE,SOURCE,ACCT29).
!SYMBOL.
OPTIONS NOBO.
:
:
```

Example 19. Prepare Off-line Compatible Binary Output File and Off-line Listing Output File.

```
!ASSIGN M:SI,(FILE,SOURCE,ACCT29).
!ASSIGN M:BO,(FILE,OBJECT,ACCT29).
!ASSIGN M:LO,(FILE,LISTING,ACCT29).
!SYMBOL.
OPTIONS LO,NODB.
:
:
```

Example 20. Binary Output for Debugging, Terminal Listing (but No LO File).

```
!ASSIGN M:SI,(FILE,SOURCE,ACCT29).
!ASSIGN M:BO,(FILE,OBJECT).
!SYMBOL.
OPTIONS TL.
:
:
```

SYMBOL ERROR MESSAGES

Symbol's error messages are output at the terminal regardless of the options selected. These diagnostics contain the same data given in standard Symbol assembly listings. Because the Teletype prints only 72 characters per line, diagnostic messages may be rearranged.

One common example of this rearrangement occurs with simple syntax errors. Three lines are printed:

1. The first line contains the assembly values (line no., hex. location, hex. contents, address classification, etc.) followed by the sequence number field (columns 73-80) of the card in error.
2. The second line is the source image, columns 1-72 of the card.
3. The third line contains error indications.

If the TL option is specified, the terminal listing will always contain the first and second lines for each source card image; the third line appears for cards in error.

Regardless of option selection, TSD Symbol always outputs an error count message at the terminal. If errors occur, this message has the form

```
EC = n
```

where n = the number of errors encountered.

When the assembly is completed, Symbol automatically returns to the TSD Executive. An assembly may be interrupted via BREAK if the user desires. In such cases the user may continue the interrupted assembly by giving the Executive command PROCEED.

6. DEBUG SUBSYSTEM

Debug is the most interactive subsystem in TSD. The user may load a program, start execution, stop it, examine items of interest, search for patterns, modify the program, continue execution, etc.

There is a continual interplay of type-in and printout in most debugging sessions. For this reason, Debug command formats are abbreviated and designed for fast interpretation by Debug. Output forms are short and simple. Nevertheless, Debug offers a high degree of flexibility and convenience.

Capabilities of the Debug subsystem are covered under the following topics:

1. Loading and starting execution
2. Stopping and continuing execution
3. Examining results
4. Modifying the executable program
5. Single-instruction execution

Users who are familiar with the on-line debugging language DDT will note much similarity with Debug. Debug's methods of operation and syntax are derived from the DDT system used on the SDS 940. Closely allied to the discussion of Debug is Section 7, "User Program Considerations".

LOADING AND STARTING EXECUTION

The user can exercise three alternatives after calling the Debug subsystem:

1. Load an assembled version of his program.
2. Create a program using Debug by entering instructions and defining symbols.
3. Evaluate quantities of interest; for example, the user could obtain the hex. equivalent of a decimal integer.

LOADING

Debug's loader can load one or more program modules. These may have been assembled by the TSD version of Symbol, by standard Symbol (off-line), or by standard Meta-Symbol (off-line). If the TSD Symbol produced the assembly, Debug permits the user to reference all nonlocal symbols. If the standard assemblers are used, only external symbols (those identified by DEF, REF, SREF) may be referenced.

;T Load

The Debug load command is ;T. For added flexibility, Debug permits the user to specify a load location (i.e., location of first word) by typing a location expression before the ;T. An example of a location expression to start loading at 100 is shown below.

Example 21. Load with Location Expression.

```
100;T
```

Usually, no location expression is given. If not, loading begins at hex. location 40 with subsequent loads following at the next available doubleword boundary (unless Debug is restarted). A maximum load, in this case, is 16,320 words. Note that the symbol table does not occupy this "user core". Symbol tables reside with the Debug subsystem of an individual user.

It is recommended that no load start prior to hex. location 40. This ensures register integrity and leaves room for TSD to save critical items.

For purposes of discussion, three kinds of loads exist:

1. Simple
2. Single-modules, multiple-file
3. Multiple-module, single file

This distinction is arbitrary. The user can make multiple-module, multiple-file loads by combining the procedures described for 2 and 3.

SIMPLE LOADING

The simple load involves one file containing one module. The user assigns that file as binary input, calls Debug, and issues the LOAD command.

Example 22. Simple Loading.

```
!ASSIGN M:BI,(FILE,OBJECT).
!DEBUG.
;T
```

If errors are detected, Debug produces error messages and lists the error severity level (if greater than zero). If this occurs, the user should consult the assembly listing. After loading, Debug outputs the next available hex. location (doubleword boundary). This information may be useful in patching the program. Assuming no errors, the Debug response to the above command might be

```
;T A3E
```

SINGLE-MODULES, MULTIPLE-FILE LOADING

The single-modules, multiple-file load involves a series of files, each containing one module. In this case, the user must reassign each file as binary input before proceeding to load it. In the following example, note that the PROCEED command is used. If Debug were called, loading would restart rather than continue sequentially.

Example 23. Single-Modules, Multiple-File Loading.

```
!ASSIGN M:BI,(FILE,OBJ1).
!DEBUG.

;T 1FE
(BREAK key is hit.)
!ASSIGN M:BI,(FILE,OBJ2).
!PROCEED.

;T A30
(BREAK key is hit.)
!ASSIGN M:BI,(FILE,OBJ3).
!PROCEED.

;T F7E
```

The advantage of single-modules, multiple-file loading is that each file may be reassembled on-line if necessary. Its disadvantage is that extra typing is required by the user.

MULTIPLE-MODULE, SINGLE-FILE LOADING

The multiple-module, single-file load requires little type-in, but if one module is seriously in error, the entire file must be rewritten (off-line). This type of load is most useful when the user has a set of routines that are fairly well checked out. In the following example, note that each ;T loads the next available module.

Example 24. Multiple-Module, Single-File Loading.

```
!ASSIGN M:BI,(FILE,PACKS,ACCT29).
!DEBUG.

;T A32
;T E44
;T 1118
;T 2FB0
```

In all cases, after an individual module is loaded, the user may examine and modify the loaded module. One important modification may be to provide values for undefined symbols. By defining the symbols before loading the next module, inadvertent link-up between modules is prevented.

If several modules contain the same defined internal symbol, Debug will only reference the symbol as defined in the most recently loaded module that contained it. As an example, suppose the routines OBJ1 and OBJ2 (see the single-modules, multiple-file load shown previously) both contain the label TEMP. Using Debug, the contents of TEMP in OBJ2 can be examined simply by using TEMP as a symbolic reference. However, some other method must be used to examine the contents of the TEMP in OBJ1. This is usually only a minor inconvenience to the user. (Note that all references to TEMP are correctly linked.)

The user is free to add patches to the end of each module of a multiple-module load. In such cases, it is the user's responsibility to start the next load at a subsequent doubleword location (i.e., by preceding the ;T command with a location expression).

The ability to examine and to patch a program prior to execution are two of Debug's most useful capabilities. Some examples will provide a preliminary understanding of these capabilities – they will be covered more fully later. In the examples below note the following:

1. Slash (/) is the command used for examining a given word.
2. Equal (=) is the command used for evaluating the last item typed.
3. Both / and = are followed by a blank. This indicates that standard formats are used: instruction format and hexadecimal format, respectively. The user may use other format options if desired; for instance, octal, integer, or EBCDIC. In such cases a format letter may be used instead of the blank.
4. Carriage return (Cr) is the command used to store in a given word. If the user types a word or instruction before the carriage return, it replaces the word or instruction displayed.
5. Line feed (Lf) is similar to carriage return in storing a word or instruction. In addition, Debug examines the next word in sequence.

The following examples illustrate these items. Note that output is underlined.

Example 25. Special Debug Capabilities.

```
ALP/ BAL,3 S3 Cr
S3/ LI,5 1 = 22500001 Cr
BET/ LW,4 S4 AW,4S4 Cr
TABL+8/ 8 Lf
TABL+9/ 9 Lf
TABL+A/ A Lf
TABL+B/ A 11 = B Lf
TABL+C/ C Lf
TABL+D/ 0 X'D' Lf
TABL+E/ 0 X'E' = E Lf
TABL+F/ 0 X'F' Cr
```

;Y Loading Complete

When the user decides that all needed modules have been loaded, Debug must be notified that loading is complete. To do this the user types the following command:

```
;Y 300A
```

Debug responds by printing the next available doubleword location after loading is complete. The unused space from this address to X'3FFF' can be used for patching. The "load complete" notification must be given prior to starting execution of the loaded program.

The ;Y command is used to instruct Debug's loader to perform the final stages of the load process. These stages are:

1. Any required Monitor Data Control Blocks (DCBs) are generated.
2. The DCB names (user and Monitor) are placed in the DCB table.

3. The Task Control Block (TCB) is created and initialized to the proper values, with allocation for the user's temporary stack.
4. Register zero is set to point to the location of the TCB.

(For a discussion of DCBs and TCBs, see the Batch Processing Monitor Reference Manual.)

Once the ;Y command has been given, no new ;Y or ;T command is allowed until Debug has been restarted. In other words, the user cannot add more load modules to a completed load.

;G Go (Start Execution)

After issuing the ;Y command, the user may start execution of his program at any desired address. The starting address is given prior to the command designator ;G. For example, if the first instruction to be executed has the label BEGIN, the user could start execution by giving the command

```
BEGIN;G
```

In general, any location expression may precede the ;G. A location expression consists of a series of symbolic labels and/or constants. These items are connected by minus signs, plus signs, or blanks. Blanks in location expressions are equivalent to plus signs; they are provided as a typing convenience (since plus signs require shifting). Symbolic labels are as defined in the Symbol language (see SDS Sigma Symbol and Meta-Symbol Reference Manual, Publication No. 90 09 52).

There are four types of constants in the Debug language:

1. Integer
2. Hexadecimal
3. Octal
4. Character (EBCDIC)

To specify EBCDIC character strings, the user types C'xxxx', where xxxx represents up to four characters not containing an imbedded single quote mark. This is the only type of constant requiring the terminating quote mark.

For the remaining constants the user may use the terminating quote mark if he wants to, but it is unnecessary. Decimal integers may be typed without any special specification, or the user may provide the integer specifier, I'. Octal numbers are preceded by the specifier, O'. Hexadecimal numbers are preceded by X' or else by ". The following equalities illustrate these formats:

```
16 = I'16 = O'20 = X'10 = "10
```

Since hexadecimal constants are so common in Sigma 7 programming, the ability to specify them by typing only one character (") is important to on-line users.

Other examples of start commands are shown below.

Example 26. Debug Go Commands.

```
BEGIN + 3 ;G
"43 ;G
"45 - "2 ;G
"40   "3 ;G
PROG + LOC - "3A ;G
O'103 ;G
67 ;G
X'43' ;G
```

Note: To recover from typing errors in a Debug command, a question mark may be typed and the command will be erased.

STOPPING AND CONTINUING EXECUTION

Once the start command is given, Debug relinquishes control to the user program. The user cannot communicate with Debug until execution stops. Three causes for execution stops are covered below:

1. An abort
2. A BREAK
3. A breakpoint

Hardware aborts, such as memory parity, are catastrophic and necessitate a complete restart of the TSD system. However, when a software abort (e.g., nonexistent instruction, memory protection violation, etc.) occurs, control automatically returns to Debug. The Debug subsystem then generates a message describing the abort, sets the instruction counter (;I) to point to the word causing the abort, and preserves the condition code existing when the abort occurred as the value of ;C. In addition, the floating controls are saved as the value of ;F (the special symbols ;I, ;C, and ;F are discussed below).

STOPPING EXECUTION

If the user pushes the BREAK key during user program execution, the program stops and control returns to Debug. To determine where execution stopped, the user may examine the instruction counter (;I) which points to the next user program instruction that would have been executed (see "Examining Results", below).

;B Set Breakpoint

Debug contains a breakpoint capability that allows the user to achieve a controlled stop at a given location. Prior to starting execution, the user gives the location at which the stop is to occur. For example, the user could give the command

```
ALPHA-3 ;B
```

which would cause program execution to stop at location ALPHA-3.

In general, any location expression may precede the breakpoint command designator (;B). Execution is stopped just prior to the given location. Debug then regains control, prints "BKPT', starts a new print line, and awaits a Debug command.

Only one breakpoint exists for a given execution; if the user gives a new breakpoint location, it supersedes a previous one. To remove a breakpoint (without replacement), the user types the command ;B without giving a location expression.

From the user's point of view, breakpoints are invisible. If the breakpoint location is examined, the original instruction is printed. However, Debug replaces the instruction (after saving it) prior to program execution. For this reason, the user is advised to avoid placing a breakpoint at a BAL or EXU, since, if these instructions are followed by a calling sequence, incorrect results occur when continuing execution. This happens because the original instruction is moved to make room for the breakpoint instruction.

CONTINUING EXECUTION

;P Proceed with Execution

After reaching the breakpoint, the user may continue execution by issuing the Proceed command ;P (see example 27).

Example 27. Debug Proceed Command.

```
BKPT
;P
```

In proceeding from a breakpoint, Debug always causes the original (moved) instruction to be executed. If the program loops back to the breakpoint location, another stop occurs.

For added flexibility, Debug permits the user to specify a count in proceeding from a breakpoint. This count (any location expression value) determines the number of times the breakpoint may be reencountered before a stop occurs (see example 28).

Example 28. Count-Controlled Proceed Command.

```
BKPT
1;P
```

The command in the above example tells Debug to allow the breakpoint location to be reached once without stopping. A second encounter would result in halting execution.

After any program stop, the user may give a ;P command to continue executing. The first location executed is always the one pointed to by the instruction counter.

;I Instruction Counter

The instruction counter, designated ;I, is initially set to the start location, given in the last module that was loaded. This value is updated whenever execution stops. The instruction counter can also be set by giving a location expression as in the following example:

Example 29. Set Instruction Counter Command.

```
LOOP-1 ;I
```

The user also may continue execution of a program by giving a new start (;G) command as discussed above. This is particularly useful when initializations are required.

Finally, a completely fresh start can be made by returning to the TSD Executive, giving the DEBUG command, and reloading. This is recommended only when a user program appears to have been modified beyond repair, since in reloading, all patches and definitions made up to that time are lost.

If control is inadvertently returned to the TSD Executive, reloading can be avoided by giving the PROCEED command. This returns control to Debug (unless another subsystem call was given) with all program code and tables intact.

EXAMINING RESULTS

Debug offers considerable flexibility in examining results. Via Debug, the user can perform pattern searching, examine the contents of a given word, or look at any number of successive words. One of several output format options can be used, including convenient default options. In addition, the user may specify what the default print options are and change them as desired.

Actual use of Debug is relatively simple because, in typical practice, only one or two commands are used frequently in examining a program, and the variations are easy to remember.

PATTERN SEARCHING

In pattern searching, the object is simply to look at words from "here" to "there" for a specified arrangement of bits. For each word satisfying the requirement, Debug prints its location and contents.

;1 ;2 Searching Limits
;L Set Searching Limits

The beginning and ending limits for a pattern search are identified by ;1 and ;2 respectively. The values for these limits may be assigned in either of the two ways shown in Example 30.

Example 30. Set Pattern Searching Limits.

```
ALPHA ;1
ALPHA + 50 ;2
```

or alternatively,

```
ALPHA, ALPHA +50 ;L
```

Either of the above limit settings could be used in searching from location ALPHA through location ALPHA + 50.

;M Set Searching Mask

The search pattern is restricted by a mask, designated as ;M. A sample mask setting command is given in Example 31.

Example 31. Set Searching Mask Command.

```
"FFFF ;M
```

The above example would be used to isolate the right half of each word searched; that is, the left half would not be examined during the search. The user is free to change the mask or search limits whenever necessary. Initially, the mask is set to all ones and the search limits are set for searching the entire user core.

;W Set Word, and Match Search

In calling for a search the user must supply a specific bit pattern for which the search is to be made. Two types of search may be used – "match" or "no-match". An example of a Match Search command is given below.

Example 32. Set Word, and Match Search Command.

```
C'ZY' ;W
```

Using the limits and mask previously explained, this example results in searching for occurrences of the EBCDIC characters ZY in the right half of each word from ALPHA through ALPHA + 50.

;N Set Word, and No-match Search

No-match search commands use ;N in place of the match search command designator ;W. The following example shows a No-match Search command:

Example 33. Set Word, and No-match Search Command.

```
0;N
```

Using the sample limits and mask previously specified, the above command would look for each word from ALPHA through ALPHA + 50 whose right half is nonzero.

In making the pattern search, both the specific bit pattern and each word examined are masked by the value specified by ;M. If the bit pattern and examined word are identical for each "one" in the mask, they are said to match. If not, a no-match condition occurs.

For each word satisfying the search conditions, Debug prints one line containing the location, a slash (/), two blanks, and the contents of the word. The prevailing default formats are used in printing the location and the contents. These formats are discussed later. A sample match search procedure is given below.

Example 34. Sample Match Search.

```
"1FFFF ;M
LOOP, LOOP + 19 ;L
ERR6 ;W
LOOP + 3/  BAL, 4  ERR6
LOOP + A/  BAL, 4  ERR6
LOOP + B/  AWM, 1  ERR6, 2
LOOP + 13/  BAL, 4  ERR6
```

The above example finds all words between LOOP and LOOP + 19 that have the location ERR6 in their address fields.

Note: If DEBUG prints out an undefined symbol it is followed by [U] to warn the user.

OUTPUT FORMATS

Debug will format three kinds of data: location values, cell contents, and expression values. In the preceding example, LOOP + A was a location value while BAL, 4 ERR6 was the cell contents at that location.

= Evaluate Expression

One example of an expression value occurs when the user requests Debug to print the hexadecimal equivalent of a location expression. This request can be made by typing the location expression followed by an equal sign and one blank; for example,

R1 + R15 - 2 = E

where R1 equals 1, R15 equals X'F', and the default option is hexadecimal. Expression values are not limited to location expressions. The following example provides the hexadecimal equivalent of an entire word.

Example 35. Expression Evaluation.

BAL, 5 R1 + R15 - 2 = 6A50000E

There are two options for printing addresses: relative, and absolute.

;A Set Address Print Format to Absolute

Initially, Debug is set to print in the relative address form, but can be changed to absolute by giving the command ;A.

;R Restore Address Print Format to Relative

To revert to the relative address format, the user types ;R.

ABSOLUTE/RELATIVE ADDRESSES

Absolute addresses are printed as hexadecimal numbers, relative addresses as symbolic except in two cases. First, if there is no label at the location in question, Debug looks through its symbol table for the nearest symbol whose value is less than the given location. If the resulting symbol is not within X'7F' words of that location, Debug prints the absolute address. Second, if the location in question is less than location X'40', Debug uses absolute addressing. This somewhat arbitrary arrangement results from a symbol conflict problem. If two or more symbols have the same value, Debug has no way of knowing which is the label attached to the location in question. Ordinarily, when this occurs, Debug uses the last entry in its symbol table (that matches the given location value). But since it is the low address values that are most frequently equated to many symbols, absolute addresses are preferred to relative addresses in this case.

Assuming that neither exception occurs, relative addresses take one of the following three forms:

1. ZORCH The symbol ZORCH has a value equal to the location in question (but may or may not be the expected symbolic address of the location).

2. ZORCH + F The nearest symbol value prior to the location in question is ZORCH. The location is 15 (X'F') words from ZORCH. In this case, the hexadecimal number shown is called the "ADDEND".

3. ZORCH + F.2 The nearest symbol value prior to the location in question is ZORCH, which is defined at a halfword boundary. Therefore, the location is 15 words and two bytes from ZORCH. If a decimal point occurs in an addend, it is followed by a byte offset number, i.e., 1, 2, or 3.

CELL CONTENTS/EXPRESSION VALUES

There are six format options available for printing cell contents. These options also are available for printing expression values. The six formats, which are defined below, are designated by the following letters: R, A, X, O, C, I. When a user gives a command to examine cell contents or to obtain an expression value, part of the command is a format letter or a blank. Blank is used when the default format is desired. For cell contents, the standard default format is R; for expression values, the standard default format is X.

;/ Set Default Format for / (Examine)

The default settings can easily be changed for user convenience. For example, to change the default form for cell contents, the user could give the command

;/C

which changes the default form to EBCDIC characters.

;= Set Default Format for = (Evaluate)

The command ;= is used to change the default format for expression values. For example, the user can type

; = I

making the default form integer constants for the "=" (Evaluate) command.

The meanings of the format letters are as follows:

R - instruction format, with relative address field.

A - instruction format, with absolute (hexadecimal) address field.

X - hexadecimal constant with leading zeros omitted.

O - octal constant with leading zeros omitted.

C - four-character EBCDIC constant.

I - signed integer constant.

The X,O,C,I formats are self-explanatory. The R and A formats both provide output in essentially a four-field format that is similar to symbolic machine instructions.

The first field is the operation code field and the second is the register field. These fields are separated by a comma.

The register field is a decimal integer. The operation code field usually contains a familiar instruction mnemonic. However, if Debug cannot translate this field into a mnemonic, it prints a percent sign (%) followed by two hexadecimal digits that correspond to the op-code encountered.

The third field is the (address) and is always preceded by a blank. If indirect addressing applies, the first character is an asterisk (*). The remainder of the third field is expressed in absolute hexadecimal form in the A-format and symbolic (relative) form in the R-format. The same exception cases occur in the relative form as were discussed under location value formatting (the same routines are used by Debug in producing the relative address and the relative location value).

The fourth field, which is the tag, is printed only if nonzero. It is separated from the address by a comma, and is expressed as an integer from 1 through 7, corresponding to an index register.

The following table relates the A and R formats to bit positions in the item printed.

Table 4. A and R Format Bit Positions

Field	Meaning	Bit Positions
1	Op-code	1-7
2	Register	8-11
3	Indirect (*)	0
	Address	15-31
4	Tag (Index)	12-14

Note: For byte or immediate type instructions, bits 12-31 are given as the address field.

In the event that fields 1 and 2 are both zero, Debug will omit printing them.

In summary, the preceding discussion has explained the Equal command, which obtains expression value. Generally, the user first types the expression to be evaluated, then an equal sign, followed by either a format letter (R,A,X, O,C,I) or a blank (for default form). The expression to be evaulated will usually be a symbol, a constant, a location expression, an instruction, or a special symbol. However, reasonable combinations of these items may also be evaluated.

There is a restriction on instruction evaluation that should be noted. Debug is designed to recognize Sigma 7 instruction mnemonics as defined for Symbol; however, Debug does not recognize Symbol or Meta-Symbol directives.

SPECIAL DEBUG SYMBOLS

Debug has a number of special symbols that can be used to provide program status and debugging status information. Some of the symbols have been discussed previously (e.g., ;I, ;1, ;2, ;M). The following table shows all special symbols:

Table 5. Special Debug Symbols

Special Symbol	Information Provided
;1	Lower limit for pattern searching.
;2	Upper limit for pattern searching.
;M	Mask for pattern searching.
;I	Instruction counter.
;C	Condition code.
;F	Floating controls.
;Q	Latest quantity typed (this special symbol is discussed in detail in remaining portions of this section).
$	Debug location counter. Points to last cell opened. The open cell may be stored into for user program modification (this is further described in a later section).
.	Synonymous with $ (provided as a typing convenience).

EXAMINE COMMAND

To evaluate any special Debug symbol, the user types the symbol, an equal sign, and a format specifier. Several of the special symbols represent locations (;1, ;2, ;I, ;Q, $, and .). The user may determine the value of these items as desribed above. In addition, the contents of the cells represented by these symbols can be examined.

/ Examine Word

To examine any one word of interest, the user types an expression followed by a slash and a format specifier. Debug responds by typing out the contents of the word addressed by the given expression. The Examine function is the most useful tool in debugging. It provides the only mechanism for looking at the program after loading or following execution.

Because of its importance, Examine has several variations and properties. One variation permits the user to dump sequential words. To do this, the user types a beginning expression, a comma, an ending expression, a slash, and a format specifier. Following is a typical example.

Example 36. Examine Command.

```
PR3,PR3+3/ LW,4 J77
PR3+1/ AW,4 CONST,2
PR3+2/ AI,4 3
PR3+3/ B DRIVER,2
```

One of the important properties of Examine concerns the Debug location counter $ (or .). The examined location becomes the value given to $. In the previous example the final value for $ is PR3+3. The user may use $ in an expression for further examination, as in

```
$+5/ LI,2 1
```

or, alternatively,

```
.+5/ LI,2 1
```

This property facilitates rapid searching for key points in programs or tables.

;Q Set Last Quantity Typed

Another important property of Examine concerns the use of the special symbol ;Q. This symbol represents the latest quantity typed out. Often, the user wants to know not only the contents of a critical location, but also the contents of the cell that the location addresses. The special symbol ;Q enables the contents to be examined.

Example 37. Examining the Location Specified in the Last Quantity Typed.

```
$+6/ LW,3 DATAX ;Q/X 9
```

In this example, the cell DATAX contained hexadecimal 9.

As a convenience, the user is not required to type the special symbol ;Q when examining its contents. The following example illustrates a case in which the user traces through a path of subroutine calls:

Example 38. Tracing – Examining Successive Quantities.

```
S+1/ BAL,4 SUB1 / BAL,5 SUB2 / BAL,6 SUB3
```

Note: When using ;Q for examination of an addressed item, only the address field is applicable; indirect addressing and indexing are not used. The following example illustrates this:

Example 39. Addressing Limitations of Examine Command.

```
SUB3/ LW,5 QUEX,3 /X 9A
```

In this example, X'9A' is the contents of QUEX, not of QUEX plus the contents of register 3. Debug examines the program as a static entity; it does not execute each location examined. Therefore, examined quantities during debugging do not necessarily reflect the path of execution the program will follow while operating.

Another common and perhaps more useful application of ;Q is its use in a program patching process. The symbol ;Q allows the user to modify the contents of the last word typed. This application is explained later in this section under "Modifying an Executable Program".

Lf (Line Feed) Open $+1 and Examine

During debugging, the user often finds one word of interest and wants to know the contents of the next word. This happens so frequently that Debug contains a special variation of Examine for looking at the next word (using the format specification given when the current location was examined).

To examine the next word, the user need only type a line feed (Lf) after the current word is printed. This is equivalent to typing a new Examine command but is much more convenient.

↑ (Up Arrow) Open $-1 and Examine

Debug also contains an Examine variation for looking at the word previous to the last one examined. For this, the user types an up-arrow (↑, shift-N on the teletype keyboard).

Table 6 summarizes variations of Examine. In the table, expr represents an expression and f represents one of the format specifiers (R, A, X, O, C, I, or blank).

Table 6. Examine Command Variations

Code	Explanation
expr/f	Examine the contents of the word at location expr using format f.
expr1, expr2/f	Examine each word from expr1 through expr2 using format f.
/f	Examine the word addressed by ;Q using format f (note that $ does not change).
Lf (line-feed)	Examine the word at $+1 using the same format used in examining $.
↑ (up-arrow)	Examine the word at $-1 using the same format used in examining $.

Note: The difference between numeric input and output may be confusing to a new user. The standard form for input numbers is decimal; the standard form for output numbers is hexadecimal. Hexadecimal was chosen as the output standard for Debug because of Sigma 7 design. Thus output closely corresponds to location values in assembly listings. However, hexadecimal is not used as the input standard because of possible conflicts with symbolic names. If hexadecimal were the input standard, the label ACE, for example, would be confused with a hexadecimal number.

At this point, many command examples of the Examine command have been given, but sophisticated combinations have not been shown. The interested user is encouraged to try any combination that offers promise since such experimentation is one of the important advantages of on-line systems such as TSD.

MODIFYING THE EXECUTABLE PROGRAM

Basically, there are two ways to modify the program after it has been loaded and optionally executed: by storing new instructions (replacements or additions) in the program, or by defining new symbolic labels (including redefining or "undefining" existing labels).

Debug permits the user to store instructions in only one location at a time and this location must be "open". There are two ways to open a location: to examine it or to give an "Open-Only" command. (The only Examine command that does not open a location is the /f command).

INSTRUCTION MODIFICATION

\ Open-Only

The Open-Only command consists of a location expression followed by a reverse slash (\). (To input a reverse slash, the user types shift-L.) The open location address is always assigned to Debug's location counter, which is identified by the special symbols ($) or (.).

Cr (Carriage Return) Store

Once a location has been opened, the user stores a full word into that location by giving an expression followed by a carriage return. The expression is ordinarily a constant or instruction, although location expressions and the value of special symbols can be stored as well. The four types of constants (decimal, hexadecimal, octal, and EBCDIC) accepted by Debug were described earlier in this section.

Instructions are formed from four basic parts:

1. Operation expression
2. Register expression
3. Address expression
4. Index expression

Though the operation expression may be any Debug expression, it is usually a Symbol instruction mnemonic (it cannot be a directive, however).

A comma is used after the operation expression, unless no register expression is desired. The register expression then follows: it is usually a decimal constant from 0 through 15.

The first blank in an instruction introduces the address expression. If indirect addressing is to be specified, an asterisk is the first character following the blank. Any location expression may be given for the address expression. A comma is used after the address expression, unless no index expression is desired. Ordinarily, index expressions are given as constants from 0 through 7.

If one of the four expressions is omitted, the stored result is equivalent to an explicit zero. The instructions in the following example help illustrate these points.

Example 40. Sample Instructions for Debug.

```
LW,14 A+2,1
LI,11 -1
B LOOP
  LOOP,6
0,0 -1
```

After the user issues a Store command (a carriage return) and it has been executed, the affected location is closed, and it cannot be stored into unless reopened. The user also can close the open location without storing by issuing a carriage return that is not preceded by an expression.

Successive instructions can be stored by using Lf or ↑ in place of Cr. After opening a given location and typing the instruction it is to contain, the user can type Lf or ↑ instead of Cr. This will store the instruction in the current location and then open the next location or previous location, respectively.

In the following example, note that the first command examines the contents of a location, thus opening it. The initial contents of each word in the patch area are shown to be zero.

Example 41. Patching Successive Locations.

```
PATCHA+10/  0 BAL,4 PRNT Lf
PATCHA+11/  0  6 Lf
PATCHA+12/  0 C'PATC' Lf
PATCHA+13/  0 C'HA  ' Cr
```

This example points out the simplicity with which a program may be patched using Debug. Using the printout, the user may check the patch at a glance, examining the instructions, their locations, and the previous contents of these locations. (Cr and Lf do not print.)

The special symbol ;Q is useful in certain common patching situations. Recall that ;Q represents the value of the last item typed. The following example indicates how ;Q can be used to minimize the typing required to change that value.

Example 42. Patching with the ;Q Symbol.

```
SBX/  B LX+2 ;Q+1 Cr
SBX/  B LX+3
```

(The second line is not necessary but shows the results of the change.)

;Z Zero Storage

For the user's convenience, Debug contains a special command ;Z for zeroing a block of locations. The following example illustrates this command.

Example 43. Zero Storage Command.

```
PATCH, PATCH+99 ;Z
```

In this example, the user zeroed 100 sequential words.

The user may zero an entire program by giving ;Z with no preceding expressions. This might be used if a new program is going to be stored one instruction at a time. In other words, the whole program is to be patched in, and the user

wants to ensure that user core is clear. For safety, Debug responds to ;Z with the message "OK". If the user wants this special Zero command to be obeyed, he confirms it by typing a period. Any other character erases the command, and user core is not cleared. An example of the special Zero command is shown later in this section.

SYMBOL MODIFICATION

As stated earlier in this section, Debug provides symbol definition commands. These commands

1. Enable the user to provide definitions for undefined symbols in a program or load module. After a symbol is defined, all previous references to the symbol are automatically corrected to reflect the given definition.

2. Allow the user to define and redefine symbols for convenience in patching a program.

Note: Once a symbol has been defined, all previous references to the symbol are replaced by appropriate values. Debug permits the user to redefine (or even "undefine") such symbols, but this does not affect program code that has already been stored. However, the change in definition will affect code that is stored after the definition change has been processed.

;U Check for Undefined Symbols

Correction of undefined symbols is a valuable and commonly used feature of Debug's symbol definition capability. Consequently, Debug contains a specific command for detecting undefined symbols – ;U. It is recommended that the user issue a ;U command after each load command (;T). If undefined symbols are found, the user will be able to provide definitions before continuing to load.

There is one situation in which an undefined symbol requires special treatment. Assume that the symbol TEMP was directed to be a local symbol (i.e., the LOCAL directive was used). Assume further that, subsequent to the LOCAL directive, TEMP was referenced but never defined within the local section.

During assembly of such local regions, Symbol/Meta-Symbol "scrubs" references to undefined symbols and replaces them with forward reference numbers, assuming that the symbols will be defined later in the local region. Consequently, when Debug processes such a program, it will detect the forward reference numbers as undefined labels but it will not be able to determine what the labels were originally. (Of course, the assembly listing will contain an error message describing the problem, but it is assumed that the user has loaded the program without reassembling.) Correcting the error involves dealing with two problems: the original symbol has been lost, and all references to it are incorrect.

Debug deals with this situation by creating a special symbolic label. If the user then provides a definition for the label, Debug automatically substitutes the definition at each location referencing the label.

When the user issues the ;U command, Debug prints all undefined symbols, including created symbols. The created symbols have the form :xxx, where xxx is a three digit hexadecimal number. Thus, :001 is assigned to the first encountered undefined local symbol, :002 assigned to the second, and so on. The user may discern which undefined local symbol (e.g., TEMP) corresponds to a given created symbol by a judicious mixture of pattern searching and examination of the assembly listing.

<< Define Symbol

To define a symbol (Debug-created or otherwise), the user types

1. A defining expression
2. A less-than sign
3. The name of the symbol
4. A greater-than sign

Examples follow.

Example 44. Defining Symbols.

```
SUBR8+5 <:001>
"1A3 <ALPH>
2 <TWO>
C'ERR5' <ERROR5>
```

! Define Symbol to have Value $

Another important Debug feature is that patching can be done symbolically rather than in hexadecimal or octal. For instance, a user can construct a subroutine while debugging, and can also store instructions that call the subroutine by name. The user must define the name, but Debug simplifies this task with an alternative method of defining labels. If the user types the name, followed by an exclamation point, Debug defines the name to have the value $. The following example illustrates the simplicity of this method of definition. The first instruction opens the location in which the patch will start.

Example 45. Symbol Definition Using Exclamation Point.

```
PATCHB+1/   0  SUBRB!_LI,2 -1 Lf
SUBRB+1/    0  AW,2  *TPOINT Lf
SUBRB+2/    0  LW,14  CONST9 Lf
SUBRB+3/    0  B  0,4 Lf
SUBRB+4/    0  TPOINT!  0,0 TABL Lf
TPOINT+1/   0  CONST9!_"9 Cr
PATCHA+10/  0  BAL,4  SUBRB Cr
```

In the preceding example, the user first defined the subroutine name SUBRB (located at PATCHB+1) and then stored the four-instruction routine. He referenced a table pointer (TPOINT) and constant (CONST9), intending to define these symbols after completing the subroutine. The user then defined and stored those two items. Note that he terminated his PATCHB insertions (see TPOINT+1) with a Cr storing hexadecimal 9, but not examining and opening TPOINT+2 (which would have occurred had he used an Lf instead of Cr). Finally, the user examined PATCHA+10 and stored a call to the new subroutine.

There are many sophisticated applications of Debug's symbol definition commands. For instance, new operation code symbols can be defined, new symbols can be defined in terms of Debug's special symbols, and redefinition can be used. Users are encouraged to try such applications.

;K Kill Symbol

Debug permits the user to "kill" symbols (make them undefined). The Kill command may be used to remove any symbol from Debug's symbol table, but this has no effect on previous references to such symbols. To kill a symbol, the user types the symbolic name followed by ;K.

Example 46. The Kill Command.

```
SUBRB ;K
```

If the above command followed the patch shown in Example 45, location PATCHA+10 would continue to reference the subroutine despite the fact that the name SUBRB would now become undefined by the ;K command.

A special application of the Kill command is useful for reinitializing Debug's symbol table. This command removes all user symbols including those obtained in loading. However, Debug retains its special symbols and the ordinary instruction mnemonic symbols (e.g., BAL, LI, AW, etc.). This command generally will be used only when the user zeros all user core assigned to him. Like the special zero command, the special Kill command for symbols requires a confirming period to be typed after Debug asks "OK". The following examples typifies use of these two commands:

Example 47. Reinitializing.

```
;Z OK.
;K OK.
```

Any response other than a period after the OK erases the command. The above application is not particularly recommended, but it does allow a user to patch in an entire program without conflicting with earlier code or symbols. (This is better accomplished by BREAKing and calling in the Debug subsystem again.)

;X Execute Instruction

There is one other command in the Debug repertoire remaining to be explained: the ;X command which executes a single instruction. The user types an expression, (usually an instruction, but any expression can be given) followed by ;X. Debug then executes, or attempts to execute, the expression as an instruction.

This command is useful for presetting registers, for testing special instruction sequences, and for checking communications between various parts of a program. The ;X command can also be used to start execution at any point in the program by executing a branch instruction to some location in the program.

Example 48. Single Instruction Execution.

```
LI,1  -5 ;X
LW,2  TEMP3 ;X
LH,3  TABLE,1 ;X
B  TCHEK ;X
```

In general, any instruction that can be stored can also be executed. Among the Sigma 5/7 instructions available to the user for debugging is the EXU instruction. This is useful in stepping through certain instructions in the existing program without having to specify their exact format.

Example 49. Selective Instruction Execution with EXU and ;X.

```
EXU  LOOP ;X
EXU  LOOP+1 ;X
EXU  LOOP+2 ;X
3/  2
```

This shows a case in which the user stepped through three instructions and then examined register 3.

Assuming that location BETA2 contains a branch instruction, the user program will start to run if the following command is given:

```
EXU  BETA2 ;X
```

This example is given as a warning to the user that the execution (;X) of an EXU instruction is not the same as single stepping at the computer console since the program can start running.

7. USER PROGRAM CONSIDERATIONS

This section discusses relationships between a user program and the TSD system. The material in this section is divided into two parts: the first part presents general information that concerns all users; the second contains descriptions of certain TSD system calls that are of use in advanced applications.

TSD RESTRICTIONS

Every user program must adhere to the following restrictions imposed by the TSD system:

1. User core is limited to 16,384 words. Of this space, the first 64 (X'40') words are reserved for registers and for TSD use. The remaining space is occupied by the user program, Task Control Block (TCB), Data Control Block (DCB) name table, and the DCB's.

2. The size of the Symbol table used in debugging is limited, at present, to approximately 2,000 symbols.

3. All input/output files for the user program must be RAD files. No other device is permitted.

4. TSD allows no more than four files to be open at one time. However, the user program may maintain a larger number by closing unnecessary files before opening new ones.

5. The Debug subsystem uses a common table for symbolic names and operation code mnemonics. The user, therefore, must avoid labels that are the same as Symbol instruction names (i.e., B, S, LI, AW, etc.).

USER RECOMMENDATIONS

In on-line debugging, the user may want to restart execution fairly frequently. For this reason, it is recommended that user programs be designed to be self-initializing. The user can then restart execution at an initialization routine to ensure that critical storage is correctly preset. Otherwise, the user may be forced to reload, which eliminates any corrections or patches made while debugging.

When the user's program occupies a single load module, he can obtain a one-to-one correspondence between the hexadecimal locations in the assembly listing and those exhibited by Debug by specifying that loading start at zero. (That is, the load command 0;T is used). However, the user must ensure that the program origin is at hexadecimal 40 or higher. For instance, the assembly might start with the following card: ORG X'40'. The resulting correspondence may simplify debugging in some cases.

In general, user programs make use of the same Batch Processing Monitor calls (i.e., CAL1's) under TSD that they would use if run in Batch mode. However, there are certain differences imposed by the TSD system. These are described in Appendix B.

For system protection, TSD intercepts all CAL1 instructions and validates them before passing them to the Batch Processing Monitor. If a call is improper (for on-line use), TSD outputs an error message, stops execution of the user program, and places the Debug subsystem in control.

SPECIAL SYSTEM CALLS

The TSD system contains service routines that may be called by the user program or subsystems. Appendix B describes these calls in detail. While a large number of these calls are for exclusive use of TSD's subsystems, five are available to the user's program. The remainder of this section discusses these five calls. This information is not required for use of TSD.

By using the TSD system calls, a user program can communicate with a terminal during execution. In other words, the program becomes an on-line program instead of an off-line program being debugged on-line. Calls are available for the following services:

1. Read one character from the input buffer.
2. Test input buffer.
3. Write one character in the output buffer.
4. Change echo control type.
5. Return control to Debug.

READ ONE CHARACTER

TSD accumulates characters from the terminal in an input buffer. To read a character from this buffer, the user program executes the CAL3,0 instruction. TSD automatically converts teletype characters (ASCII) into EBCDIC form when executing the Read call. If input data is available, TSD clears register 0 and places the EBCDIC character in its low-order byte. However, if data has not yet been entered, TSD releases the user program until data has been input at the terminal. The read call is then executed.

TEST INPUT BUFFER

The user program may test the input buffer by executing a CAL3,3 instruction. This system call sets the two low-order bits of the condition code. If the input buffer is empty, these bits are set to 00; otherwise, they become 10.

WRITE ONE CHARACTER

To write a character at the terminal, the user program executes a CAL3,1 instruction. This results in placing the low-order byte of register 0 in the output buffer. That byte should contain an EBCDIC character: TSD automatically converts it to ASCII form.

CHANGE ECHO CONTROL TYPE

Before considering how to change echo control type, two concepts must be understood — echoing and activation.

When the user depresses keys to input characters, printing of those characters does not immediately occur. All printing is under control of the computer (full duplex operation).

After receiving an input character, TSD determines whether to print it or not. When an input character is printed, it is said to be echoed. If the echo control type is zero, echoing does not occur. This control type is useful in situations where the user wants to issue data that is private, such as passwords. Ordinarily, the echo control type is nonzero. If nonzero, the echo control type allows all printable characters to be echoed. This includes letters, digits, punctuation characters, carriage return, and line feed; other control characters are nonprintable.

The echo control type also determines the activation setting. While a program is waiting for input, it is inactive and does not occupy user core. The activation setting determines the input condition required for permitting the program to become active again. The setting may cause activation on receipt of any input character or it may delay activation until an "activate" character has been input. In the latter case, TSD will accumulate data in the input buffer up to and including the activating character. Thus, the program will not become active until a complete message has been input. This type of operation increases swapping efficiency.

The following five echo control types are recognized by TSD:

Table 7. Echo Control Types

Echo Control Type	Echo and Activation Settings
0	No echoing; activate on each character.
1	Echo (all printable characters); activate on each character.
2	Echo; activate on all characters other than letters, digits, blanks, or the special characters #, :, @, $.
3	Echo; activate only on control characters, carriage returns, or line feeds.
4	Echo; activate only on carriage returns or line feeds.

The echo control type is initially set to 1 and is automatically reset to 1 whenever a subsystem is called.

To change echo control type, the user program sets register 0 to the desired type (0-4) and then executes a CAL3,2 instruction.

RETURN CONTROL TO DEBUG

This special system call permits a user program to return control to Debug. To perform this return, the user program executes a CAL3,6 instruction. By placing several of these instructions in a program, the user can force execution to stop at critical points, isolating particular areas for checkout purposes. These controlled stops are similar to a BREAK during user program execution, in that the instruction counter (;I), the condition code (;C), and the floating controls (;F) are set by Debug for user interrogation.

APPENDIX A. SPECIAL TELETYPE KEYS AND CONTROLS – MODEL 35KSR

CONTROL PANEL

Most of the control panel switches are useful only for communications or long distance computer connections. Those used in TSD are ORIG and CLR; occasionally LCL and BUZ-RLS are also used, although they are not directly related to the TSD system. A short description of each item on the control panel follows:

ORIG	Conditions Teletype set to make a call.
CLR	Clears all other keys.
ANS	Connects called station to calling station via teletype lines; called station answers.
TST	Permits test of data set from central office.
LCL	Conditions Teletype set for off-line operation.
BUZ-RLS	Silences alarm buzzer when set needs servicing.
Number Buttons	Touch-tone telephone "dial".
BRK-RLS	Lamp indicates break has been initiated by local or remote station; depressing button restores local station to sending condition.
REST	Lights up when set is transmitting to slower-speed TWX station and buffer storage of central office is nearly full.
OUT OF SERV	Lamp indicates set will not answer a call; automatic answer feature is disabled.
NORMAL-RESTORE	Rotation of switch to left puts set out of service. To restore set to service, switch is rotated to right and held until dial tone is heard. Used when replenishing paper supply.
SPKR VOL	Adjusts speaker volume for touch-tone telephone.

KEYBOARD

Most of the Teletype console keys are explained by legends printed on the key tops. However, there are three printing characters for which no legend appears:

\(backward slash) Use SHIFT-L.	](right bracket) Use SHIFT-M.	[(left bracket) Use SHIFT-K.

A brief explanation of the unusual legends follows. Except for BREAK and BELL, these controls are not used in TSD operation.

WRU	(Who are you?) Requests remote station's identification and actuates its answer-back. (CTRL-E).
TAPE	Turns on remote auxiliary tape punch. (CTRL-R).
~~TAPE~~	(Tape off) Turns off local or remote auxiliary punch. (CTRL-T).
X-OFF	(Transmitter off) Turns off local tape reader. (CTRL-S).
EOT	(End of transmission) Terminates the call and turns off both machines. (CTRL-D).
RU	(Are you?) Conditions remote station to acknowledge its own identification code. (CTRL-F).
BELL	Rings signal bell at local and remote stations. (CTRL-G).
VT	Vertical tabulation. (CTRL-K).
FORM	Automatic form feed to first printing line of next page. (CTRL-L).
RUB OUT	Null character.
LOC Lf	Local line feed (not transmitted to the computer).
LOC Cr	Local carriage return (not transmitted to the computer).
REPT	Repeat the character currently being transmitted.
BREAK	Interrupt communication (used as an "interrupt" to gain the attention of the TSD system).
HERE IS	Actuates local answer-back and generates station identification code.

APPENDIX B. SYSTEM CALLS

The TSD system allows use of certain CAL1 and CAL3 instructions (CAL2 and CAL4 instructions are not permitted).

CAL1 instructions are used to acquire various services from the Batch Processing Monitor. A subset of these CAL1's has been implemented in TSD. In general, most (RAD) I/O operations are present, but no foreground operations are allowed. A detailed list of the calls, with TSD restrictions, is given in Table B-1. (A complete "CAL1-TO-FUNCTION INDEX" appears on the inside front cover of the Batch Processing Monitor Reference Manual, Publication No. 90 09 54).

Table B-1. Monitor Function Calls for TSD.

Call	FPT Code	Function	Comments
CAL1,1	X'01'	M:REW	Allowed
	X'02'	M:WEOF	Not allowed
	X'03'	M:CVOL	Not allowed (tape operation)
	X'04'	M:DEVICE (PAGE)	Not allowed
	X'05'	M:DEVICE (VFC)	Not allowed
	X'06'	M:SETDCB	Allowed
	X'0B'	M:DEVICE (DRC)	Not allowed
	X'0C'	M:RELREC	Allowed
	X'0D'	M:DELREC	Allowed
	X'0F'	M:TFILE	Not allowed
	X'10'	M:READ	Allowed (wait is implied)
	X'11'	M:WRITE	Allowed (wait is implied)
	X'12'	M:TRUNC	Not allowed
	X'14'	M:OPEN	Allowed. The ASSIGN image, if given, overrides any DCB or FPT options. Only four files may be opened at one time.
	X'15'	M:CLOSE	Allowed
	X'1C'	M:PFIL	Allowed
	X'1D'	M:PRECORD	Allowed
	X'20'	M:DEVICE (LINES)	Not allowed
	X'21'	M:DEVICE (FORM)	Not allowed
	X'22'	M:DEVICE (SIZE)	Allowed
	X'23'	M:DEVICE (DATA)	Not allowed
	X'24'	M:DEVICE (COUNT)	Not allowed
	X'25'	M:DEVICE (SPACE)	Not allowed
	X'26'	M:DEVICE (HEADER)	Not allowed
	X'27'	M:DEVICE (SEQ)	Not allowed
	X'28'	M:DEVICE (TAB)	Not allowed
	X'29'	M:CHECK	Allowed
CAL1,2			Not allowed
CAL1,3			Not allowed

Table B-1. Monitor Function Calls for TSD (cont.)

Call	FPT Code	Function	Comments
CAL1,4			Not allowed
CAL1,5			Not allowed
CAL1,8			Not allowed
CAL1,9			Not allowed

The CAL3 instructions are used to acquire various services from the TSD system itself. Only five of these calls are available to the user program:

CAL3,0 Read character from input buffer.

CAL3,1 Write character in output buffer.

CAL3,2 Change echo control type.

CAL3,3 Test input buffer.

CAL3,6 Return control to Debug.

(See also "SPECIAL SYSTEM CALLS" in Section 7.) The remaining calls are for exclusive use of TSD's subsystems and Executive. A list of all valid CAL3's and their functions is given below.

CAL3,0 Inputs a character from the input buffer to byte 3 of register 0, clearing bytes 0, 1, and 2. (If no character is available, the user is dismissed.) No other register is changed.

CAL3,1 Outputs a character from byte 3 of register 0 to the output buffer. (If the buffer is full, the user is dismissed.) No register is changed.

CAL3,2 Changes teletype echo control type to the value specified in register 0. Table 7 in Section 7 shows the function of each echo control type.

CAL3,3 Tells the status of the input buffer. If characters are available, this call sets the condition code to xx10; otherwise, the condition code is set to xx00. No registers are modified.

CAL3,4 (Used only by the TSD Executive)
Brings a subsystem in from the RAD and transfers control to it. Registers 0 and 1 are set as follows:

R0 = Starting address of subsystem

R1 = Subsystem number:

0 = Debug,

1 = Symbol,

2 = Edit.

No other registers are changed.

CAL3,5 (Used only by the TSD Executive and subsystems)
Starts the process on the next lower level. Register 0 must contain the program status word to be used when the new process is initialized; the format of register 0 contents must be:

Bits 0-3 = Condition Code (CC)

Bits 4-7 = Floating Controls (FC)

Bits 15-31 = Instruction Address (IA)

CAL3,6 Performs a normal return to the next higher level process (same results as activating BREAK).

CAL3,7 (Used only by subsystem)
Reads page N of user's program into page M of subsystem area (user core). Note that all processes start at page 0 and go to page 31 because processes are limited to exactly 16,384 words of storage. Register 0, 1, and 2 must contain the following:

R0 = N (new page).

R1 = M (old page).

R2 = swap type

where

< 0 means that the current page is not to be written out before the new page is read in

= 0 means that the current page is written out before the new page is read in

> 0 means that the new page is not to be read in after the old page has been written out.

CAL3,9 (Used only by the TSD Executive and subsystems)
Calls the PAST or CURRENT program status doubleword (PSD) for the desired level. Whenever a process is restarted via CAL3,5 the old "current" value is saved (PAST) and the new value in register 0 is entered (CURRENT). Registers 2 and 3 must contain the following:

R2 = level number,

R3 = even (if PAST, PSD is desired),

= odd (if CURRENT, PSD is desired).

CAL3,9 returns results in registers 0 and 1 as follows:

R0 = CC, FC, IA (same as in CAL3,5)

R1 = error code

where

0 = normal return caused by BREAK or CAL3,6

1 = nonexistent instruction

2 = nonexistent memory address

3 = privileged instruction

4 = memory protection violation

5 = unimplemented instruction

6 = push-down stack limit reached

7 = fixed point arithmetic overlow

8 = floating point fault

9 = decimal arithmetic fault

10 = improper arguments to a call

11 = illegal call

12 = read error on RAD during transfer.

These error condition codes (1-12) result in corresponding messages to the user.

APPENDIX C. UNUSAL CONDITIONS AT THE USER'S TERMINAL

START-UP MALFUNCTIONS

When a terminal is turned on and BREAK is activated, the user should wait at least five seconds for a response. If nothing happens, another BREAK should be given. If no response occurs within five seconds, the system is malfunctioning (i.e., either the system is down or there are serious RAD read errors associated with the terminal's swap area). The user should abandon the terminal, try another terminal, and notify operations personnel of the difficulty.

After giving the first or second BREAK, the response may be simply an exclamation point (!) rather than the expected log-in request. This condition occurs if the previous user of the terminal left it without exiting via the BYE command. To recover, the current user should issue the BYE command, wait ten seconds, and activate BREAK. The log-in request should then occur.

EXECUTION MALFUNCTIONS

If the user's terminal becomes nonresponsive (even to the BREAK key), the system is malfunctioning. The user may try another terminal. Operations personnel should be notified of the difficulty immediately.

APPENDIX D. COMMAND SUMMARIES

TSD EXECUTIVE

Command	Explanation	Page No.
AS	RAD file assignment. Uses parameters followed by a confirming period as follows: DCB name, (FILE, file name[, acct. no.])[, (option)...]. DCB name = F:xxxxxx or M:xxxxxx file name and acct. no. are formed from a maximum of 8 letters or digits. Options allowed are: (IN), (INOUT), (OUTIN), (OUT), (REL), (SAVE), (EXPIRE, NEVER), (PASS, xxxxxxxx), (READ, ALL), (READ, NONE), (WRITE, ALL), (WRITE, NONE)	5
BY	Sign-off. Requires confirming period.	5
DE	Debug subsystem call. Requires confirming period.	6
ED	Edit subsystem call. Requires confirming period.	6
PR	Proceed with interrupted subsystem. Requires confirming period.	7
SY	Symbol subsystem call. Requires confirming period.	6

Note: Executive commands may be erased by activating BREAK prior to the confirming period.

EDIT SUBSYSTEM

Command	Explanation	Page No.
A	Append, use carriage return for next card; end text insert with C^c	11
C	Copy	11
D	Delete	10
E	End update	12
±F	FILE (take updates from command file)	16
I	Inquire update status	14
±L	List mode	14
±N	NOS. mode	15
O	Originate source file, see APPEND	13
R	Resequence	8, 12
S	Stop update	12
T	Tab set or clear	15

SPECIAL CHARACTERS

Char.	Use	Page No.	Char.	Use	Page No.
Cr	Carriage Return	8	C^c	End Text Insert	11
Lf	Line Feed	13	I^c	Tab	15
W^c	Erase	9	V^c	Literal Next	16
H^c	Effective Backspace	9	B^c	Retype	16

SYMBOL SUBSYSTEM (Options)

(See page 18 for a complete description.)

Option Form	Explanation	Page No.
BO	Binary output produced (default setting)	18
NOBO	No binary output produced	18
LO	Listing output (for off-line printing) produced	18
NOLO	No listing output (for off-line printing) produced (default setting)	18
DB	Debugging symbol table on BO file (default setting)	18
NODB	No debugging symbol table on BO file	18
TL	Listing output on the terminal	18
NOTL	No listing output on the terminal (default setting)	18

The OPTIONS response is terminated by a confirming period. To erase the option list, issue a carriage return prior to the period. A new OPTIONS request is automatically given.

DEBUG SUBSYSTEM

The Debug language contains four categories of terms: constants, format letters, special symbols, and commands.

CONSTANTS (Page 23)

Code	Meaning
Blank or I' " or X' O' C'xxxx'	Decimal integer Hexadecimal integer Octal integer EBCDIC string

FORMAT LETTERS (Page 28)

Letter	Meaning
R	Instruction format, relative
A	Instruction format, abs. hex.
X	Hexadecimal
O	Octal
C	EBCDIC
I	Integer

SPECIAL SYMBOLS (Page 29)

Symbol	Meaning
;1	Lower limit for search
;2	Upper limit for search
;M	Mask for search
;I	Instruction counter
;C	Condition code
;F	Floating controls
;Q	Last quantity typed
$	Debug location counter
.	Same as $

COMMANDS

Command	Explanation	Page No.
/	Examine word	29
\	Open Only, do not examine (Use Shift-L to type \)	31
Lf (Line Feed)	If $ was examined, open $+1 and examine	30
↑ (Up Arrow)	If $ was examined, open $-1 and examine (Use Shift-N to type ↑)	30
Cr (Carr. Return)	Store (start new line also)	31
=	Evaluate expression	27
!	Define symbol to have value $	34
<...>	Define symbol	33
;/	Set default format for / (Examine)	27
;=	Set default format for = (Evaluate)	28
;1	Set lower limit for searching	25
;2	Set upper limit for searching	25
;A	Set default format for location values to absolute hexadecimal	27
;B	Set (or clear) the breakpoint	24
;C	Set condition code	29
;F	Set floating controls	29
;G	Go (start execution)	23
;I	Set instruction counter	25
;K	Kill symbol (make it undefined)	34
;L	Set searching limits	25
;M	Set mask for searching	26
;N	Set word, and no-match search	26
;P	Proceed executing (often from breakpoint)	24
;Q	Set last quantity typed	30
;R	Set default format for location values to relative	27
;T	Load	20
;U	Check for undefined symbols	33
;W	Set word, and match search	26
;X	Execute instruction	34
;Y	Notify that loading is complete	22
;Z	Zero storage	32

To erase an incomplete command, the user types a question mark (?), see page 23.

Following is a classification of Debug commands by function. This material is given as a brief reminder of the Debug commands that are applicable in various situations. Note that duplications occur between the various sets.