SYSTEMS CONCEPTS
COMPUTER SPECIALISTS

SUITE 300 FIRST FEDERAL BLDG.
401 WILSHIRE BOULEVARD
SANTA MONICA, CALIFORNIA 90401
(213) 395-7418

TIME SHARING ASSEMBLER AND DEBUG SYSTEM

DESIGN SPECIFICATION

July 31, 1967

Mike Levitt

Block move instructions
for editor -

Newfile ← L 1-1000; L [illegible]-2600; Line1000-24
L 2601-END; FINIS.

SYSTEMS CONCEPTS
COMPUTER SPECIALISTS

SUITE 300 FIRST FEDERAL BLDG.
401 WILSHIRE BOULEVARD
SANTA MONICA, CALIFORNIA 90401
(213) 395-7418

ERRATA for TSD Design Specification

July 31, 1967

Page	Line	Shows	Change to
4	2 of !PROCEED	In cases	In most cases ✓
5	2	telling	having ✓
9	2	EBCDIC back ASCII	EBCDIC back to ASCII ✓
11	3 of second insert numbered (1)	MTS	MAT ✓
19	4 (not counting inserts)	The EDIT	EDIT ✓
	third from bottom	> b >	≥ b ≥ ✓
24	just below midpage	\ $\underline{\text{expr}}$	$\underline{\text{expr}}$ \ ✓
25	last line	quan;X	$\underline{\text{quan}}$;X

TABLE OF CONTENTS

1. DESIGN CRITERIA

The TSD system was designed with several criteria in mind. Various trade-offs were made to satisfy these requirements as well as possible while remaining within the original scope of the project. The general considerations were that is should be easy to use, simple to implement, and yet be versatile enough to handle a wide variety of jobs. Specific items considered will be discussed in the following paragraphs.

An essential feature of the system is the use of the hardware memory map. With only one user in the core at a time, the full power of the map is not required. A further restriction on the map usage results from the inability of the file management system to handle requests properly when a page boundary is crossed.

The subsystems will be written in a re-entrant manner although this fact will not be used to advantage initially. This results chiefly from having only one user in core at a time, but is also related to the restricted use of memory mapping.

Initially, the user will have a full 16K of core available. This limit was set to allow space for the remainder of the system. It is estimated that a 500 word program would be necessary to handle requests for new page acquisitions. This feature may be implemented at some future time.

The implemented scheduler is of the simple round-robin type. A speed advantage is realized as one pass through the scheduler takes a maximum of 500 μs for 8 users. This was accomplished in 30 instructions. A more sophisticated and versatile scheduler is possible to implement, but it would be longer, slower, and much more difficult to debug.

As more features are included in the executive, the size increases. Eventually swapping is required to bring in those portions of the executive which are used infrequently. This executive has been limited in scope so that swapping will not be required with a corresponding improvement in system efficiency and ease of implementation.

The DEBUG package was similarly limited with respect to the number and type of features implemented. The commands available represent those felt to be within the scope of the TSD system. Others may be added at a later time.

2. BATCH/TSD INTERFACE

The TSD System will be under the control of Batch Monitor. The 48K of core will be allocated as detailed in the TSD Planning Specification. The various programs comprising TSD are individually described in this document, and their interrelationships are explained in the Planning Specification.

Batch will pass control to the TSD Monitor at specified intervals. TSD will be allowed the same privileges as any other foreground user under Batch, except that it must be sure to return to Batch by the end of the TSD quantum. It may return to Batch before that time if the scheduler is unable to find a user who is ready to run, or if there is some RAD I/O for which it must wait. This could occur for any of the following reasons:

(1) Modifying the map to change individual pages
(2) Swapping one process for another
(3) File I/O

By returning to Batch whenever the central processor would be idle, greater system utilization is attained. If Batch were to call TSD under similar circumstances, a highly efficient system would result. does batch know this

When Batch returns control to TSD, TSD refers to a switch set by the routine which passed control to Batch so that TSD can restart itself in the proper place.

The basic quantum of both Batch and TSD will be a system parameter, but a good starting value seems to be about 100 ms. TSD uses a "watchdog" clock to make sure that is doesn't get lost and forget about the Batch Monitor. If this clock should trigger, TSD will immediately return to Batch.

A different clock is used to time each process allotted a quantum under TSD. If a process runs out its time, it is dismissed at the next convenient point. Thus, TSD has two levels of clock protection.

3. EXECUTIVE

The executive is the basic medium through which the user communicates with the "system". This communication is effected in two ways:

(1) teletype commands to the exec

(2) system calls to obtain common user services

In the initial system, the user at the terminal will be able to have access to only one of the following levels at a time:

(1) the exec teletype command processor

(2) a subsystem (EDIT, SYMBOL, or DEBUG)

(3) a user program in execution

When the user approaches a console which is not currently being used, he gets recognition from the system by pushing the BREAK button. The exec will respond with a standard entry message, and will automatically type out

!LOGIN:

The user must now type in his identification code and a dot. When it is recognized by the exec, the exec will enter the console into the system and place the user at level 1 (exec commands).

At level 1, the exec prompts the user to type in a command by starting a new line with <u>!</u>. The user then types in the first two letters of a command word. If it is illegal, the exec will respond <u>?</u> and prompt another command. If it is legal, the exec will instantly type out the remainder of the command word. For example, if he types in <u>EX</u>, the exec responds with <u>IT</u>, confirming that it recognizes the <u>EXIT</u> command. If the command requires parameters (like ASSIGN), the exec also types out a colon and expects them to be typed in. After the parameters (if any) are correctly entered, the user must type a confirming dot. The exec only obeys a command when:

(1) it is well-formed, including the parameters, if any;

(2) it is not prohibited to this user (no commands presently implemented are prohibited to any user);

(3) it is meaningful (e.g., PROCEED means nothing when no subsystem has been called yet);

(4) the confirming dot is typed.

If a carriage return is typed before a confirming dot, the command is aborted and the exec prompts another. If any of the conditions (1-3) are violated, the exec types out ? and prompts a new command.

The commands presently available at level 1 are listed below. Those characters typed by the user are underlined. See design specifications for specific subsystems for further information.

!EXIT.	Flush this console from the system - the user is done.
!EDIT.	Call in the editor subsystem (level 2). EDIT reads text from device SI (source input) and produces an updated text file on device SO (source output) by making one pass through these devices under control of directive commands from the terminal.
!SYMBOL.	Call in the assembler subsystem (level 2). The assembler reads text from device SI (source input), and assembles it in one pass, producing a listing with diagnostics on device LO (listing output) and an object program and symbol table on device BO (binary output).
!DEBUG.	Call in the DEBUG subsystem (level 2). DEBUG under directive commands from the terminal, can load an object program and its symbol table from device BI (binary input), examine and change user core, and execute the program (level 3) with or without breakpoints.
!PROCEED.	Continue a subsystem that was interrupted by a break. In most cases, this will return to requesting commands at level 2 as soon as the subsystem cleans up critical buffers and switches. In particular, EDIT has a command to inquire about the positioning of the devices SI and SO and DEBUG has commands to inspect the program counter, the condition code, and all registers in the user machine. SYMBOL merely continues assembly where it left off. To begin a new assembly, new edit, or new load, the subsystem calls SYMBOL, EDIT, and DEBUG must be used again instead of PROCEED.

!ASSIGN: device code, file name.

In the present system, the ASSIGN command is the only way to assign a device like SI or BO to a real user file like PROG1. This means that between subsystem calls, one or more ASSIGNs must be done so that the subsystem will have access to the proper files. This is open to simplification in later versions by allowing subsystem calls to have file-name parameters which are automatically assigned to the standard devices, or by letting subsystems request that information at run-time.

Programs at all levels can use CALL instructions to request certain common services. The only such services that will be available initially are listed below with their calls.

CAL3, 0	Read a character from the user terminal (input buffer), convert it to EBCDIC, clear register 0 (R 0), and place the character in the low-order byte of R 0. Nothing else in the user machine is affected.
CAL3, 1	Write the character in the low-order byte of R 0 on the user terminal (output buffer). The character is in EBCDIC code. Nothing in the user machine is affected.
CAL3, 2	Change echo control type to the integer in R 0 (see TTY routines spec). This type is initially 1 when the user logs in, and when a subsystem is called.
CAL3, 3	Skip if any characters are presently in the input buffer. Otherwise, execute the next instruction in sequence. This feature is used especially by subsystems to notice demands for attention by the user. Unlike CAL3, 0, it does not dismiss the user if the input buffer is empty.

The above calls are meaningful and are allowed at all levels. However, the remaining calls are restricted to use at certain levels. These calls have not yet been assigned specific operation codes.

Executive call	(from level 1)
CAL-, -	Call subsystem. The eight-character subsystem name is in R0 and R1. This changes levels from 1 to 2.
Subsystem calls	(from level 2)
CAL-, -	Return to exec. This goes up one level, i.e., from level 2 to level 1 in this case.
CAL-, -	Read the n'th RAD page of the user program into the m' th virtual page of subsystem core. This allows the debugger to examine the user program. All page reads from the RAD of course are preceded, if necessary, by swapping out whatever was in the real core page to be overwritten.
CAL-, -	Start up the user process. This changes levels from 2 to 3. R0 contains floating controls and CC in bits 0-7 and IA in bits 15-31. If the process exits, the exec will access floating controls, CC, and IA and put them in R0.

User process calls (from level 3)

CAL-,-	Return to subsystem normally. This goes up one level, i.e., from level 3 to level 2. It is the same as the "Return to exec" call at level 2.

4. TELETYPE INTERFACE

The TTY routines do the following:

(1) Receive a character from a user terminal and drop it into a byte array associated with that user. This array forms a ring and is called the user's input buffer.

(2) Notice characters waiting to be typed out in another byte array associated with each user and type them on the proper terminal. This array also forms a ring and is called the user's output buffer.

(3) Echo certain input characters under some conditions.

(4) Dismiss the calling process if it executes a READ (CAL3, 0) when the input buffer is empty or a WRITE (CAL3, 1) when the output buffer is full. The scheduler will reactivate the process under appropriate circumstances.

Each user's teletype interface is driven through a table. The user's table contains [in sup line [illegible], ttype] pointers to the input and output buffers, character counters, and status bits. These are:

(1) A pointer to the next byte in the input buffer into which a character from the terminal can be dropped. When the input buffer is full, characters typed in are ignored. how about echoing cr, lf.

(2) A pointer to the next byte in the input buffer out of which a character will be read by READ.

(3) A pointer to the next byte in the output buffer into which a character can be written by WRITE.

(4) A pointer to the next byte in the output buffer out of which a character will be sent to the terminal.

(5) A count of characters in the input buffer (empty=0; full=max size -- parametric, e.g., 80).

(6) A count of characters in the output buffer (analogous to (5)).

(7) The deferred echo flag. While it is ON, characters that are received from the terminal are not echoed immediately. Instead, if they are echoable, they are marked (in their high order bit) as having yet to be echoed, and left in the input buffer.

While it is OFF, characters that are received from the terminal that are echoable are echoed immediately. In any case, when a READ picks up a character that is marked as having yet to be echoed, it echoes it then. The deferred echo flag is turned ON when there is either (a) an activation condition caused by receipt from the terminal of a character that is specified as an activator by the current echo control type, or (b) an attempt is made to echo a character just received from the terminal (i.e., not demanded by a READ) but the output buffer is non-empty. It is turned OFF whenever a READ is done with one character in the input buffer (i.e., when the input buffer becomes empty).

(8) The echo-activation table. This has one byte for each character in the character set. If the byte is 0, the character is illegal. Otherwise, bit 0 tells whether the character is printable: all non-control characters are printable, Cr and Lf are printable, but all other control characters are unprintable. Bits 1-4 are not used (they are always 0). Bits 5-7 classify the character:

Class (bits 5-7)	characters
4	Cr, Lf
3	other control
2	punctuation
1	letters, digits, and blanks

When a character is received from the terminal, its echo-activation byte is fetched. If it is zero, the character is illegal, and is ignored. Otherwise, the echo control mode is examined. Echo control mode can be:

(4) All chars echo; Cr or Lf activates (class 4)
(3) All chars echo; all controls activate (class 3 & 4)
(2) All chars echo; all but class 1 activate
(1) All chars echo; all characters activate
(0) No chars echo; all characters activate

what do they echo as?
maybe as 2 char

If the mode is nonzero and the character is printable, an attempt is made to echo it (see: Deferred echo flag). A character is echoed by sending it to the back of the output buffer. Then, the character class is compared with the echo control mode; if it is greater than or equal to the echo control mode, an activation condition is present. The deferred echo flag is turned ON and a flag is set in the user's table so that the scheduler will know that activation status has been obtained. ✓

All characters in the buffers are in 7-bit ASCII code, as received from the terminal. This leaves the high-order bit free for deferred-echo marking. It also simplifies echoing, since to echo a character, it is merely transmitted unchanged (except for setting bit 0 to a 1) to the back of the output buffer. Conversion from ASCII to EBCDIC occurs when a character is sent to a process that has asked for it with a READ. Conversion from EBCDIC back to ASCII occurs only when a character is sent to the back of the output buffer from a process that has done a WRITE.

} how about xlation?

5. SCHEDULER

The scheduler scans a reactivation ring to select the next user to be scheduled. This ring is a byte array with one byte for each user (eight users in the initial implementation). Each byte contains a code which indicates the type of activation condition necessary to start this job up again.

The scheduler can be invoked either

(1) by a job being dismissed for using up its quantum, for TTY I/O, or for an abnormal termination;

(2) by Batch using up its quantum.

The scheduler scans the ring starting just after the user who was last activated. It examines each reactivation byte to determine the corresponding user's activation condition. The value of this byte is an integer indicating the following:

(0) dead job -- ignore user;

(1) unconditional start -- always satisfied;

(2) waiting for room in the output buffer -- check user's TTY table to see if output count is below threshold yet;

(3) waiting for an input activation condition -- check user's TTY table to see if activation status has been attained yet.

When a user is found whose activation condition is satisfied, his user number is sent to the swapper so that he can be brought in (Batch is allowed to run during most of the swap).

If a complete scan of all users is made and no user can be activated, then TSD will return to Batch. When Batch's quantum is used up, the scheduler is reinvoked to make another scan. Even if there is no user to activate, Batch will only be interrupted for 300-500 µs, so consequently the overhead is kept very low.

6. SWAPPER

The swapper is called by the scheduler when it finds that a particular user can be activated. Associated with the user is a level code which indicates at what level he will be running. The possible values of this code and their meanings are:

(1) The executive process

(2) A subsystem process

(3) A user process

The level code as well as all other values and tables describing user status will be kept resident in the initial implementation.

The exec process (level 1) is kept resident at all times in the initial version. However, level 2 and level 3 processes must be swapped in and out as the user changes. The swapper handles this function.

The following tables are used by the swapper:

(1) MAT (Memory Access Table). This is a word table giving control information for a virtual page of a level 2 or level 3 process. The format of an entry in the ~~MTS~~ MAT is as follows:

real page *	acc *	q	disc sector
0 7	8-9	10-11	12 31

acc = access code
q = queue (0, 1, or 2)

Those fields marked by an asterisk (*) are not used in the initial version. All pages will have the same access code (read and write). Also, the real page number can be computed by simply adding a constant offset to the virtual page number.

(2) MAP2. There is one of these byte arrays for each user's level 2 process. A pointer to the MAT entry for virtual page P of this process is kept in MAP2+P.

(3) LEVL2MAP. This is a word table. The location of the MAP2 array for user U is pointed to by (LEVL2MAP+U).

(4) MAP3. Same as MAP2 for each user's level 3 process.

(5) LEVL3MAP. Same as LEVL2MAP, but points to the MAP3s.

The swapping algorithm in the initial version is very straightforward. The current user number is compared with the next user number from the scheduler. If they are the same, the process already in core is ready to be run, so the swapper returns. Otherwise, it swaps out the current user and swaps in the next user, then returns.

Swapping out is performed by one scan through the appropriate MAP$\underline{n}$ of the current user. To swap out the virtual page P, MAP$\underline{n}$+P provides a pointer to the MAT entry in which the disc address of the page can be found. The real core page is determined by simply adding a constant offset to P. The swapper then sends the real core address and the disc address to the Batch Monitor, which adds the write request to the RAD I/O queue. After sending all the core pages used by the current user to the queue, the swapper lets Batch run until the writes are all complete. Then it scans the appropriate MAP for the next user and sends read requests to Batch Monitor in a similar manner to the earlier write requests.

All writes are completed before any reading begins to insure that Batch Monitor does not accidentally read over a page that has not yet been written out. This procedure is open to improvement in later versions of the system.

Future swappers will employ more sophisticated swapping algorithms to reduce the RAD I/O necessary to switch between users.

7. FILE I/O

All file I/O is performed by the Batch file management routines. The TSD File I/O system intercepts user I/O calls and keeps track of all device/file assignments made at the executive level. TSD makes the following checks before passing I/O calls on to Batch I/O:

(1) Referencing a device which has no file currently assigned to it will cause an error indication.

(2) Attempting to access a file that has not yet been created will cause the following action:

(a) for input files, an error indication will be given
(b) for output files, the file will be created and opened

(3) Attempting to open an already open file will cause the file to be "rewound" or positioned at the beginning of the file.

(4) Attempting to read or write an unopened file will force TSD to first open it.

TSD also augments certain executive functions as follows:

(1) When the executive process calls a new subsystem, all files for that user are closed to provide some degree of protection. However, the user must still be careful to make the correct assignments before calling a subsystem.

(2) When the executive process assigns a new file to a device, any currently open file attached to that device is first closed.

A table is kept for each user of his device/file assignments. TSD does not keep track of all files reserved under a user's ID, because the Batch system already performs this function.

8. EDIT

The TSD text editor (EDIT), under control of directive commands from the terminal, makes one pass through a file designated as source input (SI) to create the file designated as source output (SO) by either

(1) resequencing the lines from SI, or

(2) merging selected lines from SI with lines from the terminal.

Every line on a text file has text in columns 1-72 only. An eight-digit positive sequence number must appear in columns 73-80, with an assumed decimal point between columns 76 and 77. The lines in a text file must appear in ascending sequence number order.

I would encode & compre with line following ea CR

EDIT specifies a sequence number by a string of from one to eight decimal digits with an optional imbedded decimal point. There may be at most four digits on either side of the decimal point. Examples:

sequence number	col. 73-80
.0091	00000091
1	00010000
26.3	00263000
9999.8888	99998888

When EDIT is ready to accept input from the terminal, it types out at the left margin:

Lf	>	if it is awaiting a command line
Lf		if it is awaiting a line of text to be appended
Lf	@	if it is awaiting a list of columns at which to set tabs

These characters are called prompts.

EDIT features uniform editing of lines typed in, whether they are command lines, text lines, or tab-list lines. Every line is terminated by a carriage return (Cr), at which point it is obeyed if it is a command or tab-list, or appended to SO if it is a text line. Until the Cr is typed, the user can type any characters. However, the following control characters have special meanings (exactly which teletype keys will produce them will be decided later):

BS If there are any characters in the line, delete the rightmost one and type out a #. Otherwise, ring the bell.

couldn't this be done at the FT I/O level

DEL	type out @@, erase the entire line, type out Cr and the prompt awaiting a new line.
EOB	End of text block. If any characters are in the text line, append it to SO. Then type out CrLf and > awaiting a new command. This is also used while a tab-list is being typed in to abort the input and clear all tabs.
RE	Retype. Type out << Cr, then the prompt, then the line so far omitting deleted characters for better readability.
LIT	The next character typed is accepted literally; even if it is a control character such as Cr, LIT, or BS, it becomes part of the line and no special meaning is attributed to it.
TAB	Space up to but not including the next column at which a tab has been set by the last complete TABS command. If there is no such column, ring the bell instead of spacing over.

Suppose the file to which device SI is assigned has the following contents (this example will be used throughout this document):

```
--------------------col. 1-72----------------col. 73-80
A10                                          00100000
A20                                          00200000
A30                                          00300000
A40                                          00400000
A50                                          00500000
A60                                          00600000
A70                                          00700000
A80                                          00800000
A90                                          00900000   -8 digits
```

To obtain a listing of this file, the editor is called from the exec, and the following commands are used. ~~COPIESPRINT~~ (or +COP~~IESP~~RINT) switches the EDIT copy-printing mode, which is initially OFF, to ON. -CO~~PIESPRINT~~ is the command used to turn this mode back to OFF. When this mode is ON, every line written on SO that was copied from SI (rather than from the terminal) is printed on the terminal. The END command causes the unscanned portion of SI to be copied to SO and returns to the exec. In the examples in this document, that which the user types is underlined. The conversation for obtaining a listing is as follows: LIST

```
>+COPIESPRINT
>END
   10.0000 A10
   20.0000 A20
   30.0000 A30
   40.0000 A40
   50.0000 A50
   60.0000 A60
   70.0000 A70
   80.0000 A80
   90.0000 A90
```

Some commands are used to simply set tabs or switch modes. These are called "mode change" commands. The mode change commands in EDIT are COPIESPRINT, NUMBERSPRINT, and TABS. Other commands cause lines from SI or from the terminal to be written on SO. These are called "active" commands. The active commands are RESEQUENCE, DELETE, APPEND, and END. One command neither changes modes nor writes on SO; this is the INQUIRE command, which informs the user about the state of the EDIT.

All commands have the form:

$$\left\{\begin{matrix}\text{number, number}\\ +\\ -\\ \text{empty}\end{matrix}\right\} \quad \underline{\text{command word}}$$

The command word is a string of letters of which only the first is significant in determining the action to be taken. Some commands (RESEQUENCE, DELETE, and APPEND) require two sequence numbers as parameters. Others (mode change commands) may be given a sign (-=OFF, + or nothing = ON). Such parameters are provided before the command word. The comma that is shown in the above command model to separate the two numbers may be replaced by any other punctuation character except +, -, or period. However, for mnemonic reasons, colon or comma (meaning THROUGH) is generally used in the DELETE command, and slash (meaning IN STEPS OF) is used in the APPEND and RESEQUENCE commands.

All blanks are completely ignored in all commands. A command must fit in its entirety on one line. The first letter that appears on the line is assumed to be the first letter of the command word, and the rest of the line after that is ignored.

Suppose it is desired to edit the file in our example. The user makes notations on his listing something like this:

10.0000 A10
~~-20.0000 A20~~----delete
~~-30.0000 A30~~---- delete
40.0000 A40
50.0000 ~~A50~~ B50
60.0000 ~~A60~~ B60
70.0000 A70   75 B75
-------insert: 76 B76
80.0000 A80   77 B77
90.0000 A90

Since EDIT makes just one pass through SI, the user must order his commands carefully. For these changes to the file, the correct order is:

if he does: complete th edit or make a second pa

1) Delete 20-30: Use the DELETE command.

2) Replace 50-60 by new 50-60: Use DELETE followed by ~~APPEND~~. INSERT

3) Insert 75-77: Use ~~APPEND~~. INSERT

It is not necessary to tell EDIT about lines in SI that are to be kept, i.e., copied to SO. It is only necessary to tell it which lines in SI to leave out (DELETE), and what to insert from the terminal (APPEND). The conversation for this edit is as follows:

```
>20:30 DELETE
>50:60 DELETE
>50/10 APPEND
  50.0000 B50
  60.0000 B60 eob
>75/1 APPEND
  75.0000 B75
  76.0000 B76
  77.0000 B77 eob
>END
```

b:c DELETE copies lines from SI numbered less than b to SO, then skips lines from SI numbered between b and c inclusive. b/c APPEND copies lines from SI numbered less than b to SO, then appends to SO the lines typed in after the command up to EOB. The first appended line is numbered b and subsequent sequence numbers are b+c, b+2c, ..., b+nc. Finally, APPEND skips any lines in SI numbered between b and b+nc inclusive. It is recommended that to replace a range of lines with new lines, a DELETE command for the old lines precede the APPEND command for the new lines.

After the above conversation, SO contains:

```
A10                                   00100000
A40                                   00400000
B50                                   00500000
B60                                   00600000
A70                                   00700000
B75                                   00750000
B76                                   00760000
B77                                   00770000
A80                                   00800000
A90                                   00900000
```

Suppose it is now desired to resequence this file. Back in the exec, SI is assigned to the file, and a new file is designated SO. The following conversation demonstrates the use of the RESEQUENCE command:

```
>+COPIESPRINT
>100/20 RESEQUENCE
 100.0000 A10
 120.0000 A40
 140.0000 B50
 160.0000 B60
 180.0000 A70
 200.0000 B75
 220.0000 B76
 240.0000 B77
 260.0000 A80
 280.0000 A90
```

No END is necessary because RESEQUENCE automatically returns to the exec.

The two mode change commands that have not yet been explained are NUMBERSPRINT and TABS. The number-printing mode is normally ON. When it is ON, every line typed out due to a copy when copy-printing mode is ON is preceded by its sequence number, with leading zeroes changed to blanks. Also, every line that the user must type in after an APPEND command is prompted by the sequence number that will be attached to that line. This mode can be turned off by -NUMBERSPRINT.

To set tabs so that the control character TAB can be used for formatting text input lines, use the command

```
>TABS
```

This will clear all tabs, go to a new line, and prompt with an @. Then, the user types in the numbers of those columns at which tabs should be set. For example, to set tabs for FORTRAN, the conversation is:

```
>TABS
@7
```

To set tabs for SYMBOL, say:

```
>TABS
@10,19,37
```

EDIT initializes its tabs for SYMBOL program editing. To clear all tabs, say either

```
>-TABS
```

or

```
>TABS
@cr
```

~~The~~ EDIT reads from SI and writes on SO serially. The last line written on SO (copied from either SI or the terminal) is always remembered in the LAST-LINE buffer, and its sequence number is kept in the variable LAST-SEQ. Similarly, the next line to be read from SI is always available in the NEXT-LINE buffer, and its sequence number is kept in the variable NEXT-SEQ. At the beginning of the edit, LAST-SEQ is set to zero, an impossible sequence number, and the NEXT-LINE buffer is loaded with the first record from SI. At all times during the edit, NEXT-SEQ is kept just greater than LAST-SEQ by reloading NEXT-LINE from SI whenever LAST-SEQ surpasses NEXT-SEQ. This procedure assures that SO is kept in ascending sequence number order and that the files are processed serially.

At times, the user may become confused about what point in the scanning of SI and SO EDIT has reached. For example, if he presses the RUBOUT button while the edit is being performed, and then asks the exec to "!PROCEED.", it is not clear where the editor was stopped. The command

```
>INQUIRE
```

causes EDIT to type out LAST-SEQ and LAST-LINE, then NEXT-SEQ and NEXT-LINE, to resolve this problem.

? ← why do I need to know this → ?

To maintain ordered files, certain restrictions are imposed upon the numbers provided as parameters in the commands. In the following command forms, the restrictions are:

>b/i RESEQUENCE Cr	b > 0; i > 0
>b/i APPEND Cr	b > LAST-SEQ; i > 0
>b:c DELETE Cr	c ≥ b ≥ NEXT-SEQ
>TABS CrLf	
@u,v,w,...,z Cr	u < v < w <...< z

In the first three of these commands, either or both of the numbers may be omitted. In RESEQUENCE and APPEND, if b is omitted its value is taken as LAST-SEQ + i, and if i is omitted, its value is taken as 1. In DELETE, if b is omitted, its value is taken as NEXT-SEQ, and if c is omitted, it is assumed to be equal to b (i.e., only one line is deleted). For example, to append new lines to the end of a file, the command sequence that follows can be used:

```
>/5 APPEND
 285.0000 C100
 290.0000 C110
>END
```

9. DEBUG, INTERACTIVE SECTION

The basic components of the DEBUG language are:

Symbols and Constants ... 32-bit values

Forms ... letters which specify the format in which values are to be printed

Commands ... direct DEBUG

Symbols

A symbol is as defined in the SYMBOL manual, or one of the following:

$ The last register opened, whether or not still open.

The special symbols:

;M The mask used for word searches.

;1 The lower bound for all searches.

;2 The upper bound for all searches.

;Q The latest quantity typed out. Also assigned a value by "Store", "Symbol table definition", and all register-opening commands.

;C The condition code.

;I The instruction counter.

A conflict can arise if a symbolic label is spelled in the same manner as an opcode mnemonic (e.g., B). The user should avoid this situation.

Constants

Constants are as in SYMBOL: decimal digit strings and "general constants". Their values are self-defined. An alternate form for the hexadecimal X'...' is "..." The final ' or " may be omitted, except in character strings.

Forms

These letter codes are used to specify the format in which the contents of a register or any other quantity demanded by / or = should be typed.

The possible codes are:

R As an instruction with a relative address (e.g., LP+12) Hex

A As an instruction with an absolute decimal address (e.g., 2049)

X As hexadecimal digits with leading zeroes omitted

O	As octal digits with leading zeroes omitted.
C	As four EBCDIC characters.
I	As a signed decimal integer.

These form letters are reserved for possible future implementation:

B	As four bytes in decimal integer form.
H	As two half-words in decimal integer form.
D	As the doubleword decimal integer represented by the even-odd pair of locations to which this location belongs.
S	As a short floating point number.
L	As the long floating-point number determined as in D.

[illegible] ?

Expressions

Let primary refer to any symbol or constant. Let series refer to a "sum" of primaries, i.e., one or more primaries separated by '+' or '-'. Examples of series:

A

A+2-BX

$+1

;Q-1

Let a term refer to (1) a primary; or (2) a series enclosed in a single pair of parentheses preceded by BA, HA, WA, or DA. Then an expression is a term or a "sum" of terms. Examples of expressions:

byte
Half
Word
double

$-1

BA(A)+17

- but in [illegible]

Instructions

An instruction is of the form:

Command-field Space(s) Argument-field

The first blank terminates the command field. Either the command field or the argument field may be omitted, but if an argument field is present, it must be preceded by at least one blank space.

would be [illegible] that command field is terminated by space any field other del

The command field is of the form:

Operation expression, Register expression

Either expression may be zero (0). The register expression and its introductory comma may be omitted if the register expression is zero.

The argument field is of the form:

Address expression, Index expression

Either expression may may be zero (0). The index expression and its introductory comma may be omitted if the index expression is zero.

A 32-bit quantity is computed from the instruction by generating the inclusive OR of these four quantities:

(1) The operation expression.

(2) The low-order 4 bits of the register expression, shifted 20.

(3) The low-order n bits of the address expression (see below).

(4) The low-order 3 bits of the index expression, shifted 17.

indirect address

* as 1st char of arg fi

For the address expression, case (3), n is computed according to bits 1-7 of the operation expression as follows:

Operations	n
Shift	7
Immediate	20
Byte string	20
All other	17

Examples of instructions:

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

In the last example, the result will be ones in bit positions 15-31.

Commands

In the following summary of commands, these abbreviations are employed:

expr	An expression.
quan	A quantity (expression or instruction).
symb	A symbol.
spec	A special symbol (one of: ;M ;1 ;2 ;Q ;C ;I).
fmlt	A form letter (one of: R A X O C I)

form	A form letter or a blank (blank means "use default form").

Important note:

Blanks may not appear in commands except to either:

(1) introduce the argument field of an instruction;

(2) specify default form after / or =.

Symbol table definitions

expr<symb>	Define symb to have the value expr. Also assign expr to ;Q.
symb;K	Make symb undefined. If it is not a special symbol, remove it from the symbol table. ("Kill")
symb!	Define symb to have the value $ (same as $<symb>). ~~[illegible]~~
expr spec	Give special symbol spec the value expr. Example: to set the search mask, the user could say X1F77F';M
;U	Print all undefined symbols in the table.

Examine

/form note nothing preceding (input [illegible])	Examine register pointed to ;Q, i.e., the register whose location is given by the last quantity of interest (usually the last one typed out). When a register is examined, its contents are assigned to ;Q and typed out using format form. DEBUG will space before and after typing out. and open [illegible]

Open only

~~[illegible]~~expr\	Open register expr. When a register is open, and only when it is open, it is possible to change its contents using the Store command (expr Cr). The DEBUG location counter ($) is given, as its value, the location pointed to by expr. It is closed by the execution of any subsequent command except typeout (=), Examine (/form), symb!, and Delete (?). However , even when it is closed, its location remains assigned to $ until a different register is opened. When a register is opened, its contents are assigned to ;Q whether or not they are typed out.

Examine and Open

expr, expr/form

Examine all registers numbered in ascending sequence between the first and second expr inclusive. Also, open the last register examined. See the "Examine" and "Open only" commands for further details.

expr/form	Examine and open register expr.

Open adjacent

line feed	Go to a new line and type out the value of location $+1 in the current

label form mode, followed by a / and some spaces. Then open register $+1. If register $ was examined when it was opened, also examine this register using the same form. However, if register $ was not examined when it was opened, do not examine this register either.

↑	Same as line feed, but uses register $-1 instead of $+1.

Typeout

quan=form	Type the value of the expression or instruction quan in the form specified. This value is also assigned to ;Q, as is the value of any other quantity typed out by DEBUG.
=form	Type out ;Q in the form specified. This is used when a quantity has just been typed out in one form (say, instruction with a relative address) but the user would like to see it in another form (say, hexadecimal). It is also used when a register has been opened with "Open only" and not examined, but the user would like to examine it after all.

Store

quan Cr	If a register is open, assign quan to ;Q and store it in the open register. Otherwise, this is illegel. DEBUG responds to illegal commands with a ?, and ignore them. ✓
Cr	Carriage return with no quantity before it simply closes the open register if it is still open. It does not modify its contents. If there is no open register, this merely goes to a new line.

Change default forms

;/fmlt	Change the default form for "Examine-and-open" to fmlt. It is initially R (instruction with relative address).
;=fmlt	Change the default form for "Typeout" to fmlt. It is initially X (hexadecimal).

} [illegible]

Change label form mode

;R	Change the label form mode to relative. Whenever a location is typed out by DEBUG (except in the address field of an instruction typed out after / or =), it will be typed in the form symb ± constant where symb is the symbol whose value is closest to that location. If no symbol is within 100 locations, the location is typed out in absolute form instead.
;A	Change the label form mode to absolute. Locations are typed as decimal integer constants. ?

Execute and Goto

quan;X	Execute the word quan as an instruction. If it is a branch, the program

will start to run. Otherwise, control will return to DEBUG, which will CrLf and wait for a command.

expr;G	Go to location expr. The program will begin to execute from that point.

Search user core

expr;W	Word-search. Search user core between locations ;1 and ;2 and type out the locations and contents of all words satisfying the condition that their contents are equal to expr in all bit positions selected by ones in ;M.
expr;N	Not-word search. Same as word-search, but all words not satisfying the above condition are typed out.
expr;E	Effective-address search. Same as word-search, but the condition is that the effective address of the contents (assumed to be an instruction) equals expr.
expr, expr;L	Same as expr;1 expr;2 It sets both the lower and upper bounds for searches.

BKPT

Breakpoints

more breakpoint

expr;B	Set the breakpoint at location expr.
;B	Turn off the breakpoint.
expr;P	Proceed to run the program from the breakpoint. Allow the breakpoint to be encountered expr more times, then break again. A break types out CrLf() and awaits a command.
;P	Same as 0;P. Proceed to run the program from the breakpoint, but break as soon as it is re-encountered.

[illegible]

Memory loading	These commands type out "OK?" and await a confirming "." before executing. Any other response aborts them.
expr;T	Transfer a program to memory. This loads a relocatable program and its symbol table starting at location expr.
;T	This loads either a relocatable program starting at location X'100' or an absolute program.
expr, expr;Z	Zero memory. This stores zero in every location between the first and second expr inclusive.
;Z	This zeroes all of user memory.

Delete command

anything?	The partially typed command before the ? is erased. DEBUG tabs and awaits a new command.

<u>Symbol table killing</u>

;K	Erase the entire symbol table, after receiving a confirming dot as in "Memory Loading". Special symbols are not killed, and operation code mnemonics are given their standard values and restored to the symbol table.

10. DEBUG, LOADER SECTION

The stand-alone loader is to be modified and incorporated into the TSD system. The general requirements involve provisions for paging, and construction of a symbol table for use by DEBUG. The loader is invoked by the T command in DEBUG.

The DEBUG symbol table will have 4 words per entry with the following form:

w1	flag word
bit 0	1 if DEF, 0 if not
bit 1	1 if external, 0 if not
bit 2	1 if more than 4 characters in symbol, 0 otherwise ?
bit 3	reserved for expansion
bits 4-15	count of number of times this undefined symbol must have its value substituted when defined
bits 16-31	form a 16 bit flag word. If bit i=1, then pages 2(i-16), 2(i-16)+1 require substitutions when this undefined symbol becomes defined, otherwise not.
w2	value (if defined)
w3-4	symbol (left justified with trailing zeroes)

all for mem allocation

The count in bits 4-15 will be used to terminate various procedures such as searches or substitutions. The flag word will be used to minimize the number of pages accessed during these procedures.

SYMBOL does not currently provide symbol definitions or names for those symbols which are not external definitions or references. DEBUG will require this information. A modification will be made to SYMBOL which will cause these symbols and their values to be output for the loader. The loader will accept this information and include it in the symbol table for DEBUG.

The initial load operation will be quite similar to the batch load. The chief loader modification is the provision for and insertion of the paging mechanism.

The loader currently handles "simple address" forward references by means of a chaining technique. This may require that several pages be accessed to substitute the definition of a particular symbol. Even worse, a given page will be accessed once for each distinct symbol to be substituted since only one chain can be satisfied at a time. This procedure is described in the following paragraph.

Each word of user core will be represented in a bit table of 512 32-bit words (16,384 bits). This table is marked appropriately for each address substitution required. The corresponding program word contains a pointer to the DEBUG symbol table. Thus, all references of this type may be satisfied with one pass over the program, performed at the end of loading. Any bits remaining in the table represent words containing undefined addresses. The symbols required by the undefined addresses will be determined by checking the DEBUG symbol table for those symbols whose reference count (word 1 bits 4-15) is non-zero.