35P

Artificial Intelligence Project--RLE and MIT Computation Center
Memo 54--

PROPOSAL FOR A FAP LANGUAGE DEBUGGING PROGRAM

By Joel Winett

June 7, 1963

This empty page was substituted for a blank page in the original document.

Proposal for a FAP Language Debugging Program

by Joel Winett

Introduction

A time-sharing system for the 7090 computer is being developed at the M.I.T. Computation Center whereby many users can communicate simultaneously with the computer through individual consoles. In the time-sharing system a time-sharing supervisor (TSS) program directs the running of each user's program in such a manner that each user's program is run in short bursts of computation. The effect is that the user sitting at his console has complete control over his program with the unrestricted use of a large computing machine.

Through the use of commands in the time-sharing system a user who writes a program in the FAP language can assemble his program, load it into core, and start the program. In order to make the most use of the time-sharing facility the user during the debugging stages of his program will want to dynamically monitor his running program and make changes as necessary. The proposed FAP language debugging program gives the user the facility to communicate with his program using the symbols defined within his program.

Background

This is not the first effort toward developing a

language for communicating between a user and his program during the debugging stages. Every time a programmer attempts to debug his program on-line on a computer the need arises for a convenient language of communications. For those computers for which a typewriter or similiar device is used as input-output device programs have been developed to aid the user in communicating with the computer puter. For the TX-0 computer at M.I.T. the program called FLIT--Flexowriter Interrogation Tape--has been written and for the PDP-1 computer the program called DDT--DEC Debugging Tape--is used.

No symbolic debugging program has been written for the 7090 computer for two reasons: 1) the 7090 is not generally provided with a typewriter as input-output device, and 2) debugging programs are grossly wasteful of computer time. With the advent of time-sharing systems these factors are eliminated.

The Time-Sharing System

One version of the time-sharing system being developed at the M.I.T. Computation Center uses teletype units as input-output devices. The characteristics of the teletype units have been considered as prime factors in the design of a language for a FAP debugging program. The keyboard of a teletype unit contains the letters A through Z in lower case and the numerals 0 through 9 together with

the special characters and tabulation key in upper case. The keys to shift case, the space bar, the carriage return-line feed key, and the line feed only key are in both upper and lower case. The keyboards of other input-output devices, e.g. typewriters, are similiar to the teletype units; the differences lying in the special characters that are provided.

The time-sharing system uses a magnetic disc file for large scale storage, allocating a certain number of tracks to each user for storage of his programs. When a user wishes to use the system he types commands to the time-sharing supervisor program. The TSS commands provide a means of inputting a FAP symbolic program and for assembling it, producing a file containing the binary program and a file containing the table of symbols and symbol values used in the program. Other TSS commands provide a means for loading from disc storage a main program together with any additional subprograms and a means of starting the main program.

The FAP debugging program, called DEBUG, is proposed as one of the user's programs which he can load into core along with his other programs. The user can then start the DEBUG program and communicate with the computer through the DEBUG program.

The Debug Program

The DEBUG program is used to aid a programmer in

checking out and correcting his program. The DEBUG program gives the user the facility to examine registers in his program by specifying their symbolic location. The user may also examine the contents of the accumulator, the M-Q register, the index registers, and determine the condition of the lights and the state of the sense switches. The DEBUG program gives the user the ability to examine or change the contents of a register as a symbolic instruction, octal number, fixed or floating point number, an integer, or as a six character Hollerith word.

The DEBUG program provides the facility to set breakpoints and to trap or proceed from a breakpoint depending on program conditions. That is, a breakpoint program can be written to be executed at the time of the breakpoint and determine then whether to trap or proceed. The user can also assemble a program in the assembly mode of the DEBUG program, start a program at a specified location, and monitor the contents of any pertinent registers. Any changes made to the program can be recorded in symbolic form for later inclusion in the original symbolic program for reassembly. The updated assembly can be used to produce an updated listing of a FAP program.

Since a user's program is divided into a main program a and various subprograms the programmer may wish to examine registers in his main program or one of his subprograms. Thus the DEBUG program needs the symbol tables corresponding

to each program that the user wishes to refer to. These symbol tables are stored in core along with the programs. Since the user may wish to add symbols to those he has previously defined, the DEBUG program must be able to lengthen a symbol table, hence requiring more core storage. Also the user may need to make patches to his program thus requiring additional core storage for these patches. The time-sharing system provides the user with the facility of increasing the amount of core he uses. Thus the DEBUG program would request additional core storage when new symbols are defined or when free storgge space for programs is needed. The organization of core storage and the technique for expanding or compressing the amount used will be discussed in the next section.

Organization of Core Memory

During the running of the user's program his main program, subprograms, DEBUG program, symbol tables, patch programs, and breakpoint programs must all lie in core memory. Symbol tables, patch programs, and breakpoint programs are all expandable and hence the DEBUG program must organize this portion of core to accomodate various needs. Since a user gets better response when he has shorter programs, i.e. uses less core, one would like to have the flexibility of increasing the amount of core used when needed and decreasing the amount of core as the needs are

removed. Also, it is desirable to keep together areas of core used for programs but separate from the area of core used for symbol tables. The proposed scheme meets these objectives and gives the desired flexibility.

Programs are initially loaded into core beginning at some fixed lower bound. The upper bound is expandable to accomodate the needs of the user. Initially the time-sharing system loads programs into core as shown in Fig. 1, where P_i are programs and D is the DEBUG program.

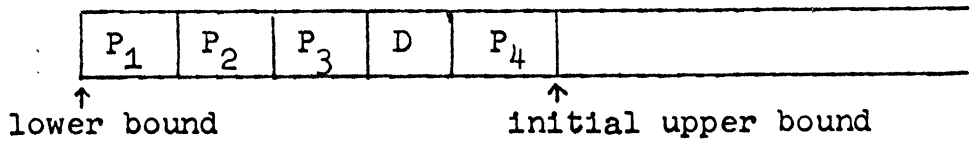


Figure 1

After the DEBUG program is started, the user issues commands to read in the symbol tables corresponding to the programs in which he wishes to refer. For example, if he wished to refer to the symbol tables S_1 and S_3 corresponding to programs P_1 and P_3 the DEBUG program requests the time-sharing system for more core space and stores these symbol tables in the area following his programs as shown in Fig. 2.

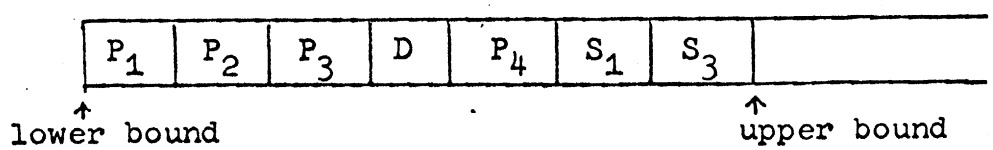


Figure 2

The symbol tables are stored in core in a list structure thus providing a simple means of adding entries to each symbol table with out requiring large blocks of core reserved for this purpose. Thus if a new symbol is defined in program P_1 an entry must be added to table S_1. The entry is stored in the core location following the last location used for a symbol, i.e. at the upper bound of core. This entry is linked to symbol table S_1 at the appropriate position.

The list structure for symbol tables does not require any additional storage space since the decrement field of the word in which the symbol value is stored is free.

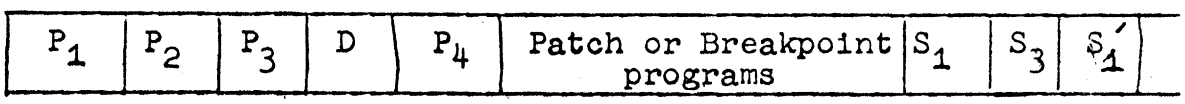
Not only are the symbols in a symbol table linked together but also the symbol tables themselves are linked to one another. Each symbol table has a header of two words. The first word in the header contains the file name of the program to which it is associated and the second word contains a link to the symbols in that table and a link to another symbol table. Thus the DEBUG program must only remember the address of the header of the first symbol table.

In addition to being list-structured the entries in the symbol tables are ordered alphabetically by the symbols providing a means of doing a partial log search for a symbol. This is possible since at the time a symbol table is initially loaded in core it is stored in a consecutive block. Only when additional symbols are added does the

symbol table get dispersed.

This organization of symbol tables does not impose any restrictions on the number of symbol tables used or on the number of symbols in a symbol table. Also, an efficient use of memory is made allowing the user to increase the number of symbols or to decrease the number of symbols. It should be recalled that (in the M.I.T. TSS system) the user gets better response when he uses less core. Thus it is desirable to kill symbols or remove tables when they are no longer needed.

Up to now no mention has been made of where additional programs, i.e. patch programs or breakpoint programs, lie in core. These programs are expandable and should not be relocated once stored. The symbol tables, on the other hand, can be relocated to make room for these new programs. Thus when patch programs or breakpoint programs are written, the symbol tables are moved down and the new programs stored following the last program loaded, as shown in Fig. 3.



lower bound upper bound

Figure 3

The user must be careful not to make changes to or store any of his data in the area of core used by the DEBUG program or the symbol tables. This kind of error can cause erractic behavior by the program. The user should periodically take a dump of core storage so that in case of erratic behavior of his program he can go back to a previous stage of his debugging.

The DEBUG Program Language

The purose of the DEBUG program is to facilitate communications between the user and his program. Consequently, in designing a language for communications it is desirable to minimize the number of characters typed by the user. Stated another way, one wants to maximize the amount of information obtained per character typed.

The experience with DDT for the PDP-1 computer has shown that a single character can be used to initiate a response. The response time for the PDP is faster than a user can type a command, hence the user gets immediate response for each character read by DDT. In the 7090 time-sharing system the situation is different; the user may have to wait up to a minute before his program is brought into core and run. In order to get the best response, the DEBUG program should be called for only when the user has typed enough information so that the DEBUG program can respond. To signal the computer that the DEBUG program should be run, a break char-

acter must be typed. The break character indicates the termination of a type-in sequence and request the time-sharing system to run the user's program. The DEBUG program will then run and produce the desired response. This procedure is in contrast to the operation of the PDP-1 system in which every character typed is a break character and requests the DDT program to read and analyze every character typed even if no response is th be made.

The time-sharing system allows the user's program to set which character or characters are to initiate a break, i.e. are to request that the user's program be run. It is suggested in this proposal that the "line feed only" character be used as the break character for the DEBUG program. This choice was made primarily on the basis that it appears desireable to have the break character exist in both upper and lower case. Another alternative is to connect a print-inch character to the line feed key. This has the objection that the change would be permanent and effect all operation of the teletype unit. In order to make the break character visible the operation of feeding a line of paper could remain connected. Additional experience with the teletype units as input-output devices will help settle this question.

Once the DEBUG program is brought into core, the characters typed are analyzed and appropriate action is taken. The characters typed on the teletype unit are read by the DEBUG program and interpreted as commands consisting of an

operator and optional operands.

The proposed language for a FAP debugging program provides the user with the ability to concatenate commands, thus increasing the amount of information obtained each time the DEBUG program is called into core. Other features of the DEBUG program allow the user to define macro command, i.e., give a name to a string of commands; and to form loops which alternately run the main program and run the DEBUH program displaying the results of a calculation during a program loop.

Command Format

The commands which the user types to the DEBUG program consists of an operator and optional operands. The operands are typed first separated by a colon, (:), followed by a space, followed by the operator. Operators are two letter symbols the first letter generally indicating the type of operation performed by the command and the second letter indicating the mode of operation. For example, the operator RS reads the contents of the symbolic location, specified as the operand, as a symbolic instruction. The mode S, for symbolic, could be O, for octal, I for integer, etc.

As indicated previously, it is desirable to type more than one command before typing the break character. The separating character slash, (/), is used to separate commands that are concatenated. The effect of concatenating

commands is the same as though the break character was typed after each command thus executing one command at a time. The feature of concatenating commands gives the user more information each time the DEBUG program is called thus giving him better response from the time-sharing system. In addition to the slash, (/), the carriage return can be used to indicate the end of a command.

The commands issued to the DEBUG program request different responses. Some commands result in printed output from the DEBUG program, e.g. the read command, RS, and other commands do not result in any printout. The user should always concatenate those commands that do not result in any printout and can often concatenate commands that do result in printed output.

Symbolic Locations

Every program or subprogram stored on the disc is given a file name, e.g. PROG1, SUBP1, SUBP2, etc. This file name is used in initially loading the program using the time-sharing commands and is also used in the DEBUG program to associate a symbol table with its program. Since a symbol may occur in many programs one may wish to explicitly indicate the program in which the symbol appears by heading the FAP symbol with the file name for the program followed by the dollar sign, ($), e.g. PROG1$LOC. The

heading of a symbolic location is not necessary if the symbol appears in only one program.

The convention is used that once a user refers to a symbol in a particular program all following symbols refer to a location in this program. If a symbol is used which is not in this program and this symbol appears in only one other program, then this other program becomes the program to which symbols are referred. Of course, a program can be explicitly indicated by heading a symbol with the file name of the program. This sets the program to which symbols are referred. If a symbol is used which does not occur in the program presently being considered and this symbol is defined in more than one other program the DEBUG program will indicate this conflict by printing an appropriate comment on the teletype unit. Thus the programmer who uses different symbols in his different programs is rewarded by not having to head a multiply used symbol.

Certain locations within the DEBUG program are given symbolic names. These locations are referred to like any other symbolic location. Thus if these symbols are used by the programmer in his program the user must head each symbol in the DEBUG program with the file name DEBUG, otherwise the

heading is not necessary.

Some of the locations within the DEBUG program that are of interest are listed below.

a. AC--a register containing the contents of the accumulator bits S,1,...,36 at the time the main program was stopped.

b. QP--a register containing the state of the Q and P bits of the accumulator at the time the main program was stopped.

c. MQ--a register containing the contents of the M-Q register at the time the main program was stopped.

d. X1, X2, X4--registers containing in the address portion the contents of the index registers at the time the main program was stopped.

e. LS--a register containing the state of the 4 sense lights, the 6 sense switches, and the 3 indicators (AC overflow, divided check, and input-output) at the time the main program was stopped.

f. BP--the location in the DEBUG program where a breakpoint program should transfer in order to proceed from the breakpoint, i.e., to pass over the breakpoint.

g. BT--the location in the DEBUG program where a breakpoint program should transfer in order to trap at a breakpoint thus saving the machine conditions and proceeding to the next debugging command.

H. FS--the name given to the first location in core which is used for patch programs or breakpoint programs.

For example, a user may store a 4 word patch for his program in location FS + 4 through FS + 7 and a breakpoint program in location FS + 10 through FS +18. If the user wishes he may give a symbol to the first location of a patch. That is, he may define location FS + 4 to be called PROG$PATCH1 and he may define location FS + 10 to be called DEBUG$BKPT1. Notice that breakpoint programs are given symbols which belong to the DEBUG program. Thus the DEBUG program has a symbol table associated with it which is stored in core along with the other symbol tables.

Meta-language

In order to present the commands available for use in the DEBUG program, a meta-language is used to describe each command in its general form. The symbols in the meta-language are the following:

a. fe--FAP expression
   A FAP expression is a string of terms separated by the operators + (addition) and - (subtraction) where a term is defined as in the FAP language except that division is not allowed.

b. fs--FAP symbol
   A FAP symbol consists of a six character symbol following the same restrictions as in the FAP language.

c. sl--symbolic location
   A symbolic location is a FAP expression or a FAP expression followed by a comma and a tag.

d. si--symbolic instruction
   A symbolic instruction consists of a FAP symbolic operation followed by a space followed by an optional symbolic location followed by an optional comma followed by an optional decrement or count.

e. fn--file name
   A file name given to a user's program.

f. n--A decimal integer.

g. wd--word
   A word is an alphabetic or numericaconstant or a symbolic instruction representing the contents of a register in core.

h. cn--command name
   The name given to the operator portion of a command.

i. com--command name
   The operand and operator portion of a command.

In describing the commands the meta-language symbol π will be used to indicate the mode of a command where π represents S for symbolic, O for octal, H for Hollerith, I for Integer, F for fixed point constant, or E for floating point constant.

Whenever a space is indicated multiple spaces can occur. Since multiple spaces and tabulation look alike on the teletype printed output they will be considered equivalent.

To represent the inoperative non-printing line feed in this memorandum the vertical bar, (|), will be used.

Commands

For each command described below the two letter symbol for the operator is given, the general form of the command, a description of the action performed by the DEBUG program, and an example showing what would be printed on the teletype printed output.

1. SL--Symbols Load

fn_1 : fn_2 : . . . : fn_n SL

This command loads the symbol tables corresponding to the programs which have file names $fn_1, fn_2, \ldots, fn_n$.

EXAMPLE: PROG1 : SUBP1 : SUBP2 SL |

2. SC--Symbols Conflict

This command prints a list of the symbols which are used in more than one program. The user can then take special precaution when using these symbols. Usually the user will head one of these symbols by the file name or redefine the symbol to a singly used symbol.

EXAMPLE: SC | SYMBOL FILE NAMES

LOC	PROG1	SUBP1	
BEG	PROG1	SUBP1	SUBP2x

3. SK--Symbols Kill

a. fn_1 : fn_2 : . . . : fn_n SK

This command erases the symbol tables from core corresponding to the programs which have file names $fn_1, fn_2, \ldots, fn_n$. Use of this command reduces the amount of core storage used by the DEBUG program thus giving the user better response.

b. If the SK command is issued without specifying any operands then all symbol tables are killed.

EXAMPLES: a. SUBP1 : SUBP2 SK |

b. SK |

4. SD--Symbol Define

sl : fs SD

This command gives the FAP symbol, fs, the value specified by the symbolic location sl. The new symbol is associated with the program in which the given symbolic location is associated.

EXAMPLE: LOC + 3 : PARAM1 SD |

5. SR--Symbols Remove

fs_1 : fs_2 : . . . : fs_n SR

This command removes the symbols $fs_1, fs_2, \ldots, fs_n$ from the appropriate symbol tables. The symbols in the operands must be used in only one program or else they must be headed with a file name.

EXAMPLE: LOOP1 : SUBP1$LOC : PARAM1 SR |

6. Rπ--Read in mode π

a. sl Rπ

The symbolic location sl is opened and its contents are printed out in the mode indicated by π, i.e., S, O, H, I, F or E. Once a location has been opened it remains open until a carriage return has been typed. One exception to this rule occurs when it is desirable to read out the contents of the address referred to in an opened register. This is done by typing a read command while a register is open, in which case the original opened register is closed and the register referred to is opened on the same line.

b. sl_1 : sl_2 Rπ

The symbolic locations from sl_1 to sl_2 inclusive are printed out in mode π. The location sl_2 remains opened at the end of this command.

c. sl : n Rπ

The n symbolic locations beginning at sl are printed out in mode π. If n is negative then the n registers beginning with sl-n+1are printed out. The last register printed out remains open at the end of this command.

d. Since the operation of opening a register is performed quite ofteh, a shorthand notation is introduced which can be used under most circumstances. The convention is that the read operator can be left out provided that the symbolic location to be opened cannot be confused with another operator. For example, LOC | will open register LOC since no confusion can arise between other commands. The programmer who refrains from using the two letter symbols that are commands can thus use the shorthand more often. When the shorthand is used, the mode of readout is the same as that of the previous read command. A command will be introduced to set the mode of readout when it is desirable to change the mode without reading any register. This shorthand notation cannot be used when a register is opened, i.e., it cannot be used to read the contents of the address referred to in an opened register.

e. To aid the user in opening consecutive registers a location sequence is kept which is set whenever a symbolic location is specified in a read command and the location

sequence is increased by one when the register is closed. The location sequence is not changed when the address referred to in an opened register is itself opened on the same line by a read command. In other words, the location sequence is changed only at the margin, i.e., after a carriage return. The symbol star (*) is used to indicated the present location in the location sequence. Thus if LOC was the last location opened, then * represents the symbolic location LOC + 1 and the user could type * RS | to open the next symbolic location in sequence. Alternately, using the shorthand notation, the user could have typed just * | .

f. A further shorthand is introduced to eliminate typing the star (*) for the location sequence. Thus, if after a carriage return just the break character is typed the location indicated by the location sequence is opened. Since commands are separated by a slash (/), successive slashes read out, in the latest mode set, succeesive locations indicated by the location sequence.

EXAMPLES: These examples illustrate what would be printed by the teletype. Some of the characters are typed by the user and others are typed by the DEBUG program. When a shorthand notation is used and an operator or operand is not typed by the user it is typed by the DEBUG program.

a) LOC RS | CLA X

(Note: register LOC remains opened.)

b. FIRST : LAST RI|
FIRST RI 432
FIRST+1 RI 827
LAST RI 107

(Note: register LAST remains opened.)

c. LOC : 3 RS |
LOC RS CLA X
LOC+1 RS ADD Y
LOC+2 RS STO Z

d. LOC | RS CLA X

(Note that the operator RS was typed by the DEBUG program. If the break character does not print a user looking at the printed output at a later time could not tell this. He does not really care since the effect is the same.)

e. * RS |
LOC+1 RS ADD Y

f. |LOC+2 RS STO Z

(Note that the user closed register LOC+1 by a carriage return and opened register LOC+2 by typing the break character only.)

g. LOC+1 RS | ADD Y RI | 26
|LOC+2 RS STO Z RI | 55

(This example illustrates opening a register referred to in the address part of an opened register. Location Y contains the integer 26 and location Z contains the integer 55 after the STO instruction. Notice that the location sequence was not ch changed by reading location Y.)

7. Mπ--Mode set to π

This command sets the mode for reading registers to π.

EXAMPLE: TABLE RH | PARAMS
MI/* |
TABLE+1 21

(Notice the use of the star for the location sequence and the example of concatenating commands.)

8. Eπ--Equals in mode π

This command prints the contents of an opened register in the mode indicated by π. The command has no effect if a register is not opened.

```
EXAMPLE:  TABLE : 2 RH |
          TABLE RH       PARAMS
          TABLE+1 RH     000000A     EI |     21
          T
```

9. EA--Effective Address

sl EA

This command writes the value of sl as a symbolic location. This command is useful when the symbolic location contains a tag.

```
EXAMPLE:  Y,2  EA |      LOC+32
```

10. Dπ--Deposit word in mode π

a. sl : wd_1 : wd_2 : . . . : wd_n Dπ

This command deposits the words $wd_1, wd_2, \ldots, wd_n$ into successive locations beginning at location sl. The mode of interpretation of the words is indicated by π.

b. wd Dπ

The deposit command can be issued after a register has been opened. In this case the word in the operand is to be stored in the opened register with the register remaining open.

c. A shorthand convention can be used to eliminate typing the deposit operator when used to change the contents of an opened register provided that the variable field of the word cannot be confused with a command operator. This

shorthand notation for the deposit operator is used only when a register is open.

EXAMPLE: a. SUM : CLA Z : ADD Y : STO Z DS /
Y : 13 DI |

b. LOC RS | CLA X CLA Y DS |

c. LOC RS | CLA Y CLA Z |

11. Aπ--Accumulator in mode π

This command reads the contents of the location AC and QP in the DEBUG program printing the contents of AC in the mode indicated by π and the state of the Q and P bits. To change the contents of the AC, the user must first read its contents and then change it using the deposit command. The P and Q bits cannot be changed using this command.

EXAMPLE: a. AI | QP = 01 AC = 2304

b. AS | QP = 00 AC = CLA X CLA Z DS |

12. QP--read Q and P bits of accumulator

This command reads out the contents of register QP in the DEBUG program which can then be changed if desired. This is the only way these bits can be set by the user. Note that the user is not allowed to modify any portion of the DEBUG program except by first opening the register in question.

EXAMPLE: QP | QP ≠ 01 11 DO |

13. Qπ--read MQ register in mode π

This command reads the contents of the location MQ

in the DEBUG program printing its contents in the mode indicated by π. To change the contents of the location MQ in the DEBUG program the user must first read its contents and then change it using the deposit command.

EXAMPLE: QS | MQ = ALS 2

14. X1, X2, X4--Index Registers 1, 2, and 4

These commands read the contents of the corresponding register in the DEBUG program. The mode of printed output is an integer. The contents can be examined in a different mode by using the Eπ command. To find the 2's complement of the index register the command EC, Equals Complement, is used.

EXAMPLE: X1/EC |

X1 = 32601 EC 167

(Note that by concatenating commands before executing the break character the information required is obtained during a single call of the DEBUG program.)

15. LS--Lights and Switches

This command prints the state of the sense lights, sense switches and indicators.

16. PS--Program Start

sl PS

This command starts the users program at the symbolic location sl.

EXAMPLE: BEG PS |

17. PA--Program Assemble

fn PA

This command prepares the DEBUG program to read lines of a FAP symbolic program typed at the teletype. The DEBUG program then requests sufficient free storage space from the time-sharing system and assembles the FAP program. The program is stored after the last register used as free storage, thus moving all the symbol tables up in core to make room for the patch or breakpoint program.

The symbols defined in this program are added to the symbol table associated with the program with file name, fn. If the program being assembled is a breakpoint program then the symbols are associated with the symbol table with file name DEBUG. The symbolic program terminates with the FAP pseudo operation END. The only other pseudo operations that can be used are PZE, BSS, and BES.

The FAP program to be assembled must not use any other of the FAP pseudo operations. Thus each line or FAP instruction is assembled into a single register in core. Details of the assembly program will not be discussed here.

If the user is assembling a patch program he must insert the necessary transfer instructions in his main or subprogram. The responsibility lies with the user to keep track of his patches. This reqirement does not appear to be burdensome to the user as reported by users of the DDT and FLIT debugging program.

To give the user some help in keeping track of his patches and the other changes that he makes in his program, the DEBUG program makes a copy of the changes and patches on a file on the disc. This file contains in symbolic form all changes the user makes to his program using the deposit command and all patches assembled using the PA command. This file can be printed out using the time-sharing commands to remind the user of his changes and patches and to produce in an organized manner a printed copy of patches and changes. Using this symbolic file an auxilliary program could update the user's symbolic program to be reassembled in the time-sharing system.

EXAMPLES: a. PROG1 PA |

```
PATCH1   ADD X
         ADD Y
         STO Z
         TRA ALPHA
Y        PZE 1
         END |
```

(Note that the above patch program uses the s symbols X, Z, and ALPHA which have previously defined in PROG1

b. DEBUG PA |

```
BKPT1    CLA     PROG1$Z
         TZE     DEBUG$BT
         TRA     DEBUG$BP
         END |
```

(In this breakpoint program the symbols BT and BP refer to the locations in the DEBUG program to which the program transfers to proceed from a breakpoint restoring the machine conditions or the location to which h

the program transfers to trap at the breakpoint. Recall that if the symbols BT and BP are not used in the user's programs then the header can be eliminated. Also, if the symbol Z occurs only in PROG1 then this header is not required.

18. BD--Breakpoint Define

sl : fs BD

This command inserts a breakpoint in symbolic location sl. The method of inserting breakpoints is to store an STR FAP code in the operation portion of the symbolic location sl. The original contents of the operation field is saved in the DEBUG program. This allows the user's program to alter the address and decrement portion of the location in which a breakpoint was inserted. When an STR trap occurs control is transferred to the DEBUG program.

At the time of the breakpoint the machine conditions are saved and the DEBUG program transfers to the location indicated by the FAP symbol fs. The FAP symbol fs is a symbol associated with the DEBUG program. It could be the location BT at which the DEBUG program traps or it could be the symbol indicating the first instruction of a breakpoint program.

EXAMPLE: a. ENDLOC : BT BD |

b. Z+1 : BKPT1 BD |

19. BR--Breakpoint Remove

sl_1 : sl_2 : . . . : sl_n BR

This command removes the breakpoints located at symbolic locations $sl_1, sl_2, \ldots, sl_n$ restoring the operation field to its original value.

EXAMPLE: BKPT1 : BKPT3 BR |

20. BK--Breakpoint Kill

a. fn_1 : fn_2 : . . . : fn_n BK

This command removes all breakpoints in the programs with file names $fn_1, fn_2, \ldots, fn_n$.

b. If the BK command is issued without specifying any operands then all breakpoints are removed.

EXAMPLE: a. SUBP1 : SUBP2 BK |

b. BK |

21. BL--Breakpoint List

This command lists all breakpoints that have been defined.

EXAMPLE: BL |

ENDLOC : BT

Z+1 : BKPT1

22. BP--Breakpoint Proceed

n BP

This command is issued after a breakpoint has occurred and request the DEBUG program to continue running until n breakpoints have occurred regardless of where they occur and to trap after the n-th breakpoint. At the time of the trap the DEBUG program prints the instruction location counter as a symbolic location.

EXAMPLE: 1000 BP |

ILC = ANSLOC

23. Concatenation of Commands

com_1

23. Concatenation of Commands

$com_1/com_2/.../com_n$

Commands to be executed in sequence can all be typed before typing the break character by separating the commands with the slash (/). After the break character is typed each command will be executed in sequence during a single call of the DEBUG program. This feature allows the user to get more running time for each break character typed and is preferred over executing each command separately. By concatenating commands the user can set up a sequence of pperations where a change is made to his program, the program is started and upon completion results are printed out.

EXAMPLE: PARAMS : 10 : 15 : 20 SI/ BEG PS/RESULT : 3 RI |

24, CD--Command Define

"$com_1/com_2/.../com_n$" : cn CD

This command gives the user the ability to name a string of commands. The name cn given to the string of commands can be any combination of two letters or numbers which are not names of commands in the DEBUG program. The user chooses the command name as he pleases. Every time the user wishes to execute the string of commands named, he types the command name, cn, which he gave to the string.

This feature of the DEBUG program is useful when a user wants to examine the same locations in his program many times.

EXAMPLE: "ANS1 RI/ANS2/TABLE : 3 RH/MASK RO" : C1 CD |

C1 |

ANS1	RI	23
ANS2	RI	1963
TABLE	RH	ABLE
TABLE+1	RH	BAKER
TABLE+2	RH	CHARLY
MASK	RO	011111011111

(Note that in the second command the operator is not specified. It is assumed to be a read command in the mode I, i.e. RI.)

25. CR--Command REpeat

"$com_1/com_2/.../com_n$" : n CR

This command provides the user with a way of repeating a series of commands a specified number of times. Each time the command sequence is executed the count number n is decreased by one until the count is zero. Using the repeat command, CR, a user can alternate between running his program and executing debugging commands. For example, a user may wish to monitor his program as itsis going through a program loop. He may wish to print out the contents of a location which contains a number which is calculated using a series expansion. To monitor the convergence of this number the user would like to print out its value after every 10 approximations, and he might like to do this 5 times.

The following example illustrates the setting of a breakpoint at the answer location, starting the program, and performing the desired debugging loop.

```
EXAMPLE:  ANS : BT BD/BEG  PS/"10 BP/ANS RI" : 5  CR |
               ILC = ANS
               ILC = ANS
          ANS RI       3055
               ILC = ANS
          ANS RI       3066
               ILC = ANS
          ANS RI       3059
               ILC = ANS
          ANS RI       3064
               ILC = ANS
          ANS RI       3061
```

Summary

The language for a FAP debugging program presented here attempts to reflect the characteristics of 1) the FAP language, 2) the M.I.T. time-sharing system, and 3) the teletype input-output device. An attempt has been made in the design of this communications language to make debugging an easier task. Experience with previous debugging programs has shown that much can be done to improve man-machine communications. The development of time-sharing systems promises to be a big step in making computers more accessible and in minimizing the time spent programming, running, and checking out a program.

In considering the system characteristics of the debugging program the line feed character on the teletype was chosen as the break character because it exists in both upper and lower case. The organization of core storage separates the area for programs and the area used for symbles

bol tables. The list structure organization of symbol tables allows symbols to be added or deleted making an economical use of storage. This is an important factor since it effects the response time from execution of a command to performance of an action in the time-sharing system.

One of the important features of the proposed FAP language debugging program, in regard to improving response time, is the ability to concatenate commands. Thus by concatenating commands the user reduces his immediate demands on the time-sharing system but takes full advantage of the speed of the 7090 computer to do a lot of computation when the DEBUG program is being executed. Other features of the proposed debugging language give the user flexibility and are powerful debugging aids. These include:

a. the ability to symbolically debug more than one program at a time, i.e., to have available the symbol tables of many programs.

b. the ability to define macro commands, i.e., give a name to a sequence of commands.

c. the repeat command for defining debugging loops which alternate between running the user's programs and executing debugging commands.

d. an improved facility for defining breakpoints based upon breakpoint programs which can be used to dynamically monitor a running program.

e. an assembly mode for making extensive modifications to a program in the form of patches. The assembly mode can also be used to program directly in the debugging language or to write breakpoint programs.

In regard to the command structure, consistency and ease of expansion should be of prime importance. A language full of exceptions is not easy to learn or use and becomes difficult to improve or expand.

The implementation of the proposed debugging program should be begun immediately and should take full advantage of all the features of the time-sharing system. As the time-sharing system becomes operational the users of the system will demand better communication languages. These demands, though imposing, should not influence the designers to compromise in making a useful and powerful debugging aid.

Acknowledgement

The author wishes to express his appreciation to Mr. Tom Hastings for his helpful suggestions, thoughts, and comments. Many of the ideas presented in this memo were suggested by Mr. Hastings and others grew out of the numerous discussions which we had.

References

1. "An Experimental Time-Sharing System," F. J. Corbato, M. Merwin-Daggett, R. C. Daley; Proc. of the WJCC, May 1961; p. 305.

2. "DDT," Memo PDP-4-1, PDP-1 Computer, Dept. of Elec. Eng.,

References

1. F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, "An Experimental Time-Sharing System," Proc. of the WJCC, May 1961; p. 305.

2. "DDT", Memo PDP-4-1, PDP-1 Computer, Dept. of Elec. Eng., M.I.T., Cambridge 39, Mass., February 15, 1962.

3. "FLIT--Flexowriter Interrogation Tape: A Symbolic Utility Program for TX-0," Memo M-5001-23, TX-0 Computer, Dept. of Elec. Eng., M.I.T., Cambridge 39, Mass., July 25, 1960.

CS-TR Scanning Project
Document Control Form

Date : 11/30/95

Report # AIM-54

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 36 (48 IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☒ Other: COPY OF MIMEOGRAPH (POOR)

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-36) UN# TITLE PAGE, UN#ED BLANK, 1-34
(37-40) SCANCONTROL, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 11/30/95 Date Scanned: 12/7/95 Date Returned: 12/7/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 12/7/1995

M.I.T. Libraries
Document Services