ARPA ORDER NO. 189-1

MEMORANDUM
RM-6027-ARPA
AUGUST 1969

AN ON-LINE DEBUGGER FOR OS/360 ASSEMBLY LANGUAGE PROGRAMS

W. H. Josephs

PREPARED FOR:
ADVANCED RESEARCH PROJECTS AGENCY

The RAND Corporation
SANTA MONICA • CALIFORNIA

ARPA ORDER NO. 189-1

MEMORANDUM
RM-6027-ARPA
AUGUST 1969

AN ON-LINE DEBUGGER FOR OS/360 ASSEMBLY LANGUAGE PROGRAMS

W. H. Josephs

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of ARPA.

The RAND Corporation
1700 MAIN ST. • SANTA MONICA • CALIFORNIA • 90406

PREFACE and SUMMARY

To the assembly-language programmer, the ability to debug a program on-line can mean a significant reduction in his overall debugging effort. Many such on-line systems are available for use, notably DDT and its derivatives for the PDP computers; however, none is available for the system 360 operating under OS, where dumps and off-line traces must be used. This Memorandum describes such a system, DYDE (Dynamic Debugger), that provides the programmer with the DDT-like capabilities of breakpoint insertion, modification of data and code, and symbolic-label references. It can operate under any option of Operating System 360, using only a small, user-written "pingpong" SVC, and communicating with the programmer using either the 1052 operator's console (employing WTO and WTOR) or, preferably, a 2260 graphics console (using the OS Graphics Access Method support).

The work covered by this Memorandum is sponsored by the Advanced Research Projects Agency. It is one of several fundamental tools to be used at Rand and externally to significantly increase the production of programmers and programming.

CONTENTS

I. INTRODUCTION

The environment provided by the multiprogrammed options of Operating System 360 is not the most suitable for debugging. It is primarily a batch system, with a programmer's card deck disappearing into the card reader and reappearing at some future time on a printer. What happens in between is often impossible to discern; any attempt to monitor a program's execution (e.g., the setting of an address stop) is so complicated that it is nearly impossible. In this environment, debugging is difficult--at the conclusion of a program, the programmer either has successful execution or some indication of program error. If he planned ahead (and was lucky), his output will include not only an indication of the actual error, if one occurred, but trace information (either through OS TESTRAN facilities or his own printouts) to help him determine the problem. However, he is usually presented with a dump, containing a numerical reference to the completion-codes manual. More importantly, the dump represents the state of the system when OS decided it could not continue the program's execution; the user must discover why it went wrong by educated guesses and by "playing computer" with his program. The difficulty and sheer wastefulness of this procedure is extremely evident. For this purpose, an on-line symbolic debugger can be invaluable.

One traditional environmental requirement for on-line debugging is an on-line system with remote job-entry capabilities and file-management functions, or a dedicated machine and its operator console. DYDE (Dynamic Debugger), the system described herein, was developed in and for the former environment using the RAND Simultaneous Graphics System. However, the debugger can be used in a normal OS batch environment using any available 2260 graphic-display terminal or even the on-line operator's typewriter.

The text that follows includes an external description, including invocation procedures and command formats, followed by a brief explanation of the internal operation of the debugger (including the "pingpong" SVC).

II. DYDE

INVOCATION OF *DYDE*

DYDE is executed as an OS job using a standard set of Job Control Statements (see Fig. 1). These define the library in which DYDE resides (JOBLIB), a library containing the program or programs to be debugged (SYSLIB), and a scratch file for organizing the symbol table (SYSUT1). In addition, any JCL statements defining data sets that are used by the program to be debugged must be included (in this context, DYDE contains a facility for overriding both the SYSLIB and the SYSUT1 DD names if the program being debugged needs them). Figure 2 illustrates a procedure for assemblying, link editing, and debugging. In any of these procedures, as soon as DYDE receives control, it writes out a message indicating its readiness for user commands.

```
//      JOB
//JOBLIB  DD  library definition
//S1  EXEC  PGM=DYDE
//SYSLIB  DD  library definition
//SYSUT1  DD  UNIT=SYSDA,SPACE=(TRK,(5,1))
//SCOPE  DD  UNIT=Ø4Ø
```

Fig. 1--Sample JCL for Invocation of DYDE Using the 2260 Version

```
//     JOB
//JOBLIB  DD  library definition
//STEP1  EXEC  ASMFCL,PARM.ASM='TEST',PARM.LKED='TEST'
//ASM.SYSIN  DD  *
    source deck
/*
//STEP3  EXEC  PGM=DYDE
//SYSLIB  DD  DSNAME=*.STEP1.LKED.SYSLMOD,DISP=(OLD,DELETE)
//SAMPLEDD  DD  data set description
//SYSUT2  DD  UNIT=SYSDA,SPACE=(TRK,(5,1))
//SCOPE  DD  UNIT=Ø4Ø
```

Fig. 2--Sample Assemble, Link Edit, and Debug JCL

Note: The SYSLIB card points to the output of the Link Edit step, and the user will override (using the *DDNAME command) the SYSUT1 default name with SYSUT2.

DEVICE DEPENDENCIES

DYDE can interact with the user through either an IBM 2260 display station or the IBM 1052 operator's console. For this purpose, two versions of DYDE exist; one for the 2260 interaction, the other for the 1052 (described in Appendix A, p. 21). Because these devices are extremely different, the mechanics of the interaction differ significantly. However, the basic operations are the same.

The more natural mode of operation, and the one for which DYDE was originally designed, uses the IBM 2260

graphics-display station. This is an alphanumeric device with a CRT capable of displaying up to twelve lines of text; each line can contain a maximum of 80 characters. The control unit for the 2260, the IBM 2848, buffers typed messages, displays typed characters, and handles display regeneration and cursor advancement. The main CPU is presented with an attention interrupt only when the enter key is depressed. The OS Graphics Access Method (GAM) schedules an asynchronous routine of DYDE that, in turn, activates the main routine in DYDE. The message is then read and acted upon.

The twelve-line screen face is divided into two logical sections:

1) The first three lines--0, 1, and 2--are for DYDE-user communication;
2) The remaining nine lines--3 through 11--are for data display.

Data are written in the second area in a wrap-around fashion--the first data item is displayed starting on line 3, the next on line 4, and so on until the screen is full. At this point, new data are displayed starting again on line 3 (erasing automatically the previously displayed data); and line 4 is erased, providing a visual delimiter between old data and the most recent display. Each new line of data display is handled in this manner, with the data over-writing the oldest data on the screen, and the next numbered line blanked as a delimiter.

The three remaining lines--0, 1, and 2--are used for command processing. The user enters his commands on line 2 beginning with a start symbol (displayed as ▶ and usually written automatically by DYDE) followed by the command; this is followed by the attention or the enter key (displayed as ■) that interrupts the CPU. DYDE reads the message and immediately echos (i.e., rewrites) it on line 0.

This provides not only positive verification of the transmission but also, as the user prepares to type the next message, a useful indicator of the last operation performed. Any data display requested is displayed on the first free line of the data area, and the line following is blanked. Finally, DYDE writes a confirmation message on line 1 and prepares line 2 for the next command by erasing it, writing the start symbol, and positioning the cursor at the first free space. Should the command be syntactically incorrect, an error message is written on line 1--the echo message on line 0 provides the user with ready reference for discovering his error--and the data region of the display is not disturbed.

The discussion that follows is concerned primarily with the 2260 version of DYDE rather than the 1052. Significant differences will be noted; however, all command and message formats, as well as operational details, are described for the graphic station version rather than for the typewriter version.

TYPICAL DEBUGGING SESSION

A typical debugging session begins when DYDE gains control and writes its READY message. At this point, the user can identify the program to be debugged, perhaps overriding one or more of the DD names that DYDE normally uses. After the program has been successfully LOADed, the full spectrum of DYDE commands is available to the user. He may indicate to DYDE that he wishes execution of his program to be temporarily suspended when control reaches specified locations; this is done by inserting breakpoints at these locations. Commands exist for modifying parts of his code or his data. He could then request DYDE to begin execution of his program. At this point, four events can suspend program execution and transfer control to DYDE:

1) Control reaching a previously defined breakpoint;
2) Executing the pingpong supervisor call as an assembled instruction in the user's program (e.g.,

useful when debugging an overlay program when a particular load is not originally in core);

3) An asynchronous interrupt from the user at his 2260 (not available for 1052 users);

4) The program program checks (e.g., it specifies an invalid address or operation code).

For release 17 of the operating system, a fifth event can suspend program execution:

5) Whenever the user's program is terminated abnormally by the operating system.†

At any of the above halting points, the user may, for example: 1) display data in his program, 2) modify data, instructions, or register contents, 3) create hardcopy of specified areas within his program, 4) insert new breakpoints, or 5) delete old breakpoints. He may resume execution of his program from the point at which it last halted (the "current" breakpoint) in either the instruction step mode (execute one instruction at a time) or in the uncontrolled mode, in which case only one of the above events can suspend program execution again. In this manner, the user can watch his program's execution to catch an error as it is occurring as well as test his program with sample data or temporary patches.

DYDE COMMANDS

The available commands that the user may issue fall into two general categories: 1) those that create the proper environment for debugging the program, and 2) those that cause actual data display from the program.

All "environmental" commands begin with an asterisk, followed by the command keyword. If parameters are necessary,

†Items two through five are considered by DYDE to be implicit breakpoints.

the keyword is followed by an equal sign; then the parameters are entered and delimited by one of several special characters (the selection of the special characters is made by the user). These special characters include the following symbols: ' ', '_', ':', ';', '/', '.', '$', and '@'. In the command descriptions that follow, the '/' is used. Most of the commands allow different forms of the parameters; however, each legal form is stated explicitly, and no other form may be used. Within the parameter descriptions, the user substitutes the indicated quantity for lower-case items and supplies the operand exactly as shown for upper-case items. Several commands contain a quantity called "loc" as a parameter. In general, this refers to a location within the user's program. Its actual use is described at the end of this section.

The commands (with the preceding start symbol and the trailing, end-of-message symbol omitted) follow.

1) `*NAME=pgmname`

defines the linkage-editor-assigned member name of the program to be debugged. This program is LOADed from the data set defined by the SYSLIB DD card (or any overrides--see *DDNAMES command, p. 9). While LOADing the program, the debugger organizes the symbol table, if present, and writes it out on the data set defined by the SYSUT1 DD card (also overridable--see the *DDNAMES command, p. 9). The command may be issued at any time; if a previous program is in core, it is deleted, and the debugger reinitializes itself before LOADing the new program.

2) `*FINISH`

terminates the debugger.

3) `*PARM=parameter information`

sets up pointers so that the information following the equal sign is passed to the program according to normal OS standards.†

4) `*DDNAME=syslib/sysutl/sysprint`

causes the debugger to override, in its DCBs, the default name for the library data set, the symbol-table, the utility-work data set, and the data set to contain hardcopy output. The normal names are SYSLIB, SYSUT1, and SYSPRINT. However, as indicated previously, the user may need these names for his program's execution. In this case, he may, using this command, override one, all, or any combination of these three names; e.g., if the user included a DD card name PRIVLIB instead of the SYSLIB card, he would issue *DDNAME=PRIVLIB. If he needed the name SYSUT1 and SYSPRINT for his program's execution, he could include DD cards named A and B and issue the command *DDNAME=/A/B. To be effective, this command must be issued before the associated data set is needed; to issue a *NAME command followed by the *DDNAME would be meaningless unless the user wished to debug two programs from two different libraries.

5) `*SETMODE=[NEXT = {ON / OFF}]`

causes the debugger to change its global mode setting. NEXT=ON tells the debugger to recognize the next *GO (or

†If the parameters are coded PARM='XYZ' on the EXEC card, the command should be *PARM=XYZ.

a null command) as a command to execute the next instruction; in this way, the background program can be run one instruction at a time. NEXT=OFF resets this.

6) `*TRACE`

causes DYDE to print the current contents of the screen face into the SYSPRINT data set and, thereafter, to print each displayed line. If DYDE is tracing currently, *TRACE turns off tracing.

7) `*PRINT`

requests the debugger to copy everything displayed currently on the 2260 screen face into the SYSPRINT data set (this name is overridable--see the *DDNAME command, p. 9). In this way, the user may keep a history of his debugging sessions and also develop a hardcopy trail of errors for later analyses. This command does not exist in the 1052 version.

8)
```
(1)  *BREAK=name
(2)  *BREAK=name/DEL
(3)  *BREAK=/DEL
(4)  *BREAK=name/loc
(5)  *BREAK=name/loc/verify string
```

instructs the debugger to insert a breakpoint (cases 1, 4, and 5) or delete a breakpoint (case 2 and 3). In the former case, a breakpoint, with the given name, is inserted at a specified location In case 1, it is inserted at the last displayed position; in case 4, at the named location; and in case 5, at the named location--after DYDE has verified that the supplied string (in hex) matches the information that is actually in core at that location. If the two strings

do not match, the location is displayed, but no breakpoint is inserted nor is any other change made. Case 2 tells the debugger to delete the named breakpoint; and case 3 tells the debugger to delete the current breakpoint (if one exists).

9) `*GO`

instructs the debugger to execute (or resume) the current program. If this is the first *GO issued after an *NAME, the program begins at the link-editor-assigned entry point. If the program is halted currently at a breakpoint, control is resumed at the breakpoint's location unless an *RESUME has modified this address. If the program has program checked (a specific type of 360 interrupt such as an invalid address specification), the only way to resume it without reloading a fresh copy is through the *RESUME.

10) `*RESUME=loc`

specifies that when program execution is restarted, the debugger should resume execution at the specified address rather than starting at the current breakpoint. This is the only way to resume a program that has program checked. Note that great care must be exercised when using this command to guarantee that registers and program cells are properly set so that another program check does not occur.

11) `*DUMP`

tells the debugger to dump itself and the program as if an ABEND (an abnormal termination SVC with the code of 100) were located at the current breakpoint rather than the machine instruction actually there.

12)
```
(1)  *MODIFY='COND'/value
(2)  *MODIFY=loc/value
(3)  *MODIFY=# reg no/value
(4)  *MODIFY=value
(5)  *MODIFY=loc/rep value/verify value
```

instructs the debugger to modify the program being debugged. In cases 1 and 3, the debugger modifies either the condition code set when the program resumes or the value of the specified register. For the condition code, the user supplies the mask as if he were testing it--*MODIFY='COND'/8 would cause the instruction BC 8 to branch, whereas BC 7 would not. For the register, the hex digits supplied replace the same number of digits in the register--if register 3 contains ABCD1234 and if the command *MODIFY=#3/0000 were issued, the new value would be 00001234. In case 2, the specified location is modified by the supplied value; in case 5, the specified location is modified by the rep value, after comparing it with the verify value; and in case 4, the last displayed location is modified. All hex digits supplied are modified in all cases; i.e., if location 1000 contained 47F0,1234 and if the command *MODIFY=47AF were issued, the new value would be 47AF,1234. Note that in cases 2 and 4 the value supplied may contain imbedded commas.

13)
```
(1)  *CSECT=loc
(2)  *CSECT
```

defines a new context for the evaluation of expressions used for the loc parameters. In case 1, the location specified is used as the new base. Case 2 resets the program's base to the first byte of the load module.

Several previous commands contain a location specification as a parameter (signified by loc in the command's syntax). Wherever this is required, the user may code the sum or difference of any combination of the following elements:

1) ?hex value--hex displacement from the current base point (see *CSECT);
2) &hex value--absolute displacement from the first addressable byte in the machine;
3) Decimal value--decimal displacement from the current base point (see *CSECT);
4) *--location of the current breakpoint;
5) # followed by a register (i.e., #3);
6) Character string--absolute location of the specified symbol;
7) Any sum or difference of the above enclosed in parentheses (no limit on the depth)--meaning the contents of the expression within the parentheses.

Cases 1, 2, and 7 require further explanation. When the program is loaded initially, all displacements are evaluated with reference to the first byte of the load module. This is independent of the linkage-editor-assigned entry point. Thus, ?44 refers to 68 (decimal) bytes after the first byte of the load module. The *CSECT command may be used to modify this; i.e., if an *CSECT=?44 is issued, the reference to ?44 refers to a location 136 (decimal) from the entry point. In this way, the user may move from one control section to another without having to compute displacement plus linkage-editor-assigned, control-section address. This feature may be used when, for example, one program dynamically loads another. The user may plant a breakpoint just before the actual transfer of control, discover the location of the entry point of the LOADed program (it should be in a register), and plant a breakpoint there (perhaps using the *BREAK=/(#15) command). When the second breakpoint is reached, the user may issue a *CSECT=* command to set the context to the LOADed program.†

†In this case, the symbol table is unavailable for the LOADed program.

Examples of valid loc parameters follow:

1) (((&10))+4) would locate the current TCB (location x'10' in the machine that contains the address of the communications vector table; the first word points to a double word in core, and the second word contains the address of the current TCB).
2) If register 3 contained the value x'10', ((((#3)))+4) would accomplish the same thing. If cell CVTLOC in the user's program contained the value x'10', ((((CVTLOC)))+4) would also locate the current TCB.
3) SAVE+4 should specify a location 4 bytes after the symbol SAVE.
4) (#15) would specify the location pointed to by register 15.

The other general category of commands requests displays of items or status about the program being debugged. These do not begin with an asterisk followed by a keyword, but are merely commands that specify what is to be displayed. These commands follow:

1)

(1)	'R'
(2)	# followed by register number

requests the debugger to display either the contents of all registers (case 1) or only the specified register (case 2). For the 2260 version of DYDE, either one line is written for a single register display or four lines, each containing the contents of four registers, are written. For the 1052 version, case 1 calls for writing three messages to the operator (without reply) for registers 0 through 11 and one WTOR (which forms the basis for the next command) for the remaining four registers. In either case and for either version, the registers are displayed as they were at the

last breakpoint, including any subsequent manual modification (or all zero if the program has not yet begun execution).

2) `'COND'`

requests a display of the current-condition code as a decimal value between 0 and 8; i.e., if the condition code is displayed as 8, a BC 8 will branch but a BC 7 will not.

3) `'BREAK'`

requests a display of the current breakpoint information. All data regarding currently active breakpoints are displayed as well as identification of the current breakpoint.

4)
```
(1)  loc
(2)  loc/length
(3)  loc//modifier
(4)  loc/length/modifier
```

causes the display of a particular location (see the loc parameter discussion above), and defines a 'current' location to be used if the next *MODIFY or *BREAK does not specify an explicit one. If no length or modifier information is supplied and the loc specification contains no symbol, a 4-byte hexadecimal value is displayed. If a symbol is present, its length and type attributes are used. A length, which must be a decimal less than 32, determines how many digits will form the final display. The modifier may be C, B, or R, or it may be omitted. If C is coded, the value will be displayed as characters; B requests the display as a bit string of ones and zeros; and R requests a display relative to the current base point. However, if R is qualified by some value in parentheses (e.g., loc//R(BASE2)), the displayed value is relative to the value of BASE2.

One other command to DYDE exists: the asynchronous interrupt to the user's executing program. After a user has indicated his desire to resume execution of his program, DYDE does not receive control again until another breakpoint is encountered. However, if the user provides an asynchronous interrupt (by simultaneously depressing the enter and shift keys on the 2260), DYDE is given control by OS, interrupting the program being debugged (which is currently executing). DYDE plants a breakpoint where the program will resume and then terminates interrupt processing. When OS resumes the program, this breakpoint is executed, and DYDE is entered. In this manner, the user, after requesting resumption of his program, may interrupt it from the console and use all of DYDE's facilities.

SYMBOL TABLE

To allow the user to make symbolic references to his program, DYDE uses the OS TESTRAN facility to provide a symbol table. The assembler's test option tells it to provide the symbol table as part of its output object module. Similarly, the linkage-editor's test option tells it to write a composite symbol table (a concatenation of each symbol table present in the input load or object modules) along with the load module. Under normal processing this symbol table is ignored; i.e., when a load module is brought into core, the symbol table is stripped off. However, before loading a program in response to an *NAME command, DYDE checks the disk data set containing the program for a symbol table. If the load module on the disk does not contain symbol-table entries, it is simply loaded into core, and the user is informed that symbols are not available.

However, if symbol-table entries are present, they are read into core; an index is built through a hash technique; and they are written into the SYSUT1 data set. Each symbol used is present along with its attributes of type and length

and its displacement. The composite, external symbol dictionary (CESD) of control sections, produced by the linkage editor, is used to build a map of the program so that each symbol may be assigned an address relative to the load point rather than a displacement from its control-section origin. As each symbol is retrieved, the first four characters are multiplied by the last four, and the middle seven bits of the resulting 64-bit product are used to index a 128-entry hash table. Each table entry contains an index to a block of data on external storage and a displacement within that block. All symbols with the same hash entry are chained together, each pointing to the block and displacement of the next symbol. Each block contains enough space for 200 symbols; the most recently referenced block is kept in core to minimize disk accesses. This method seems to work efficiently for the on-line user expecting rapid response.

USER SVC

One major deficiency of the 360 hardware, which any debugging system must overcome, is the requirement that any transfer of control be accompanied by the setting (and the destruction) of one of the sixteen general-purpose registers. Thus, the transfer of control from the debugged program at breakpoints cannot be accomplished merely by a branch, but must be performed by an instruction that is independent of register settings. The most likely candidate is a supervisor call (SVC) and its associated supervisor call routine, which can arrange for saving all sixteen registers and the transfer of control. However, the modification of the user's program when such an SVC is inserted to represent a breakpoint requires that destroyed instructions be executed interpretively out of line, if the breakpoint is to be used in the future. This is quite

expensive since approximately 120 instructions are in the 360 repertoire, and each one's interpretation must be coded separately. Using the EXECUTE instruction to execute the one modified instruction out of line is another possibility. However, this requires that all sixteen registers be properly set before the execute instruction is issued, and that control be transferred to the next instruction in the program without destroying any register contents.

To solve this problem, DYDE employs a type III user-written supervisor call (SVC) that allows both DYDE and the program to be debugged to reside as "co-routines" in the same job. This SVC can be viewed from the outside as having a ping-pong effect on the control flow. Each time the SVC is issued, after an initial call, control is passed to the other co-routine; i.e., the first call passes to the SVC routine address within DYDE for register-and program-status-word (PSW) save areas, one for itself and one for the program being debugged. Thereafter, each issuance saves the registers and PSW of the issuing co-routine in its area and restores the registers and PSW of the other member of the pair. Thus, each breakpoint inserted in the program being debugged calls for DYDE to lift and save the current instruction at that location and to plant the two-byte SVC. When the SVC is executed, control passes to DYDE at an entry point specified by it; a note is made of the location where the SVC was issued. When the user indicates he wants his program resumed, the lifted instruction is moved into a special area in DYDE; the program's resume address is updated to point to this location; and DYDE issues the SVC. This causes the program's registers to be restored and control to be passed to the lifted instruction. If it is a branch, control passes directly back to the program. However, if it is not a branch, control will pass to the next instruction in this special area, which happens to be another ping-pong SVC call. Since it was issued while the

program was in execution, control is passed to DYDE, which notes that the SVC was issued from within its own address space and that the lifted instruction dropped through. DYDE then calculates the address of the instruction following the lifted instruction, places it in the program's resume-program-status word, and reissues the SVC. This causes control to return to the program, which remains in control until another breakpoint is reached (see Fig. 3, p. 20). The only instructions that cannot be executed when moved are the Branch and Link and the Branch and Link Register, which are location dependent--they load a specific register with the current contents of the location counter and then branch to another location. DYDE interprets both instructions.

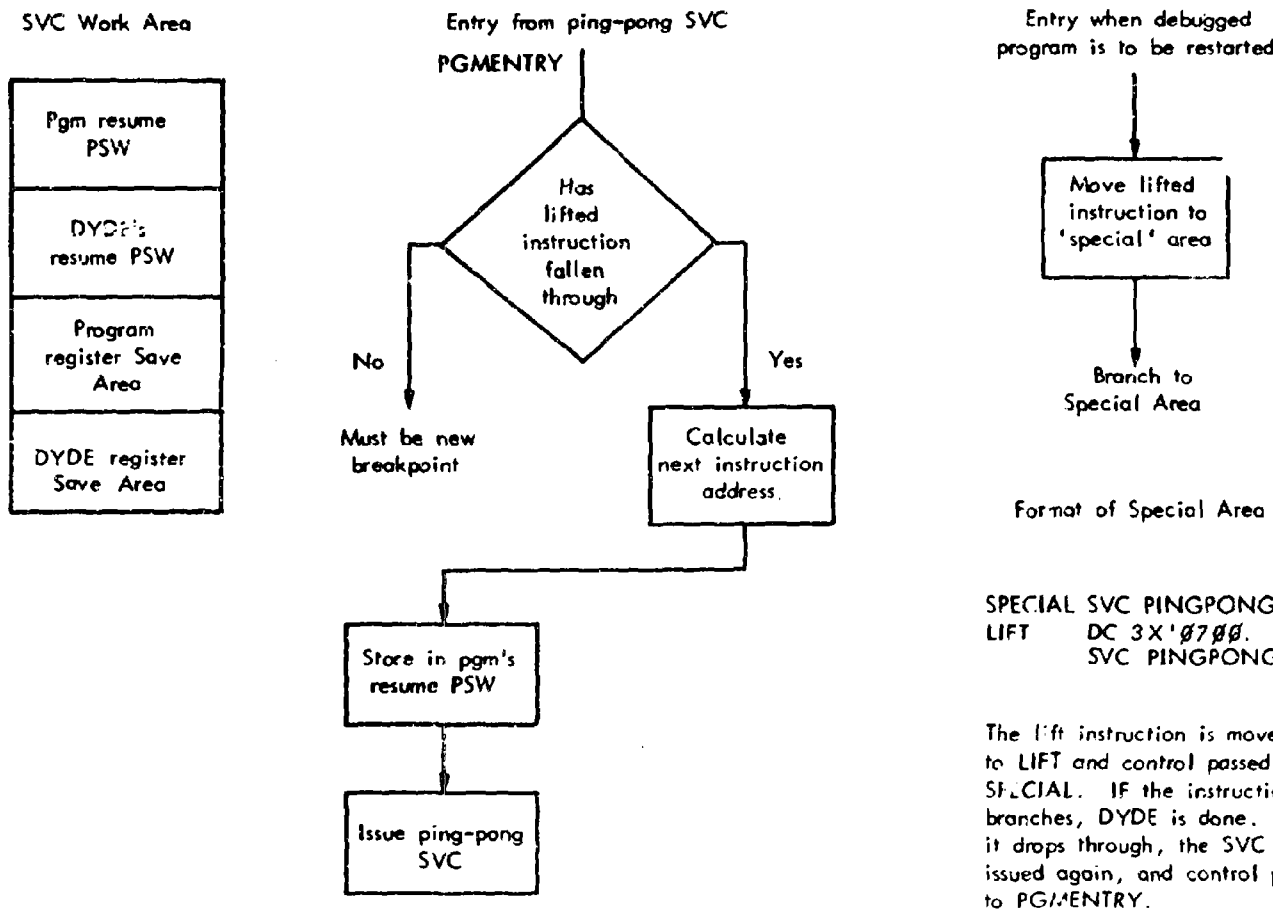

Fig. 3--User Program--DYDE Interfaces

Appendix A

1052 OPERATION

The 1052 is the normal OS operator's console. DYDE uses the Write to Operator (WTO) and the Write to Operator with Reply (WTOR) facilities to communicate with the user. These macros allow any program to type a message on the typewriter, or to type a message and wait for a reply. This facility provides a very rudimentary form of interaction; not only is the typewriter slow, but the form of user commands is, of necessity, burdensome. More importantly, the console is used by OS for communications with the operator. As such, it types out not only declarative but informative messages and expects some replies. Thus, a user wishing to use DYDE on a 1052 must tolerate other console activity; separate those messages sent to him by DYDE from other operator messages, usually by noting the message content; and tag his commands with the number of the message to which he is replying.

The mechanism for these replies is bothersome. The user first depresses the REQUEST key; then, when the system responds with the proceed light, he must type the character R (short for REPLY), leave a space, and then type the following: 1) the number of the outstanding message to which he is replying, 2) a quote, 3) the message body, 4) a terminal quote, and 5) the end of block. Assuming the user has received a proceed light, and is replying to message 3, he must type:

```
R 03,'THIS IS AN EXAMPLE.'
```

followed by an end of block.

Using this operation, DYDE initially types out a READY message and waits for a reply. The user responds to this message using the reply mechanism--by issuing a legal

command, and being careful to note the number (or tag) associated with the READY message. DYDE responds to each request with a message. If the request requires more than one line, at least one WTO is issued, with no wait for reply; it is followed by a WTOR and a wait for reply. In this manner, DYDE can debug a program that resides as one of many jobs in a multiprogrammed environment, and still keep the interference with normal system operations at a minimum.

Appendix B

COMMAND ABBREVIATIONS

The following command abbreviations are available:

Abbreviation	Full Form
*NA	*NAME
*M	*MODIFY
*BR	*BREAK
*FI	*FINISH
*DD	*DDNAMES
*CS	*CSECT
*RE	*RESUME
*TR	*TRACE
*S	*SETMODE
null command (i.e., just the enter symbol)	if mode is next, then *NEXT if mode is not next, then *GO

DOCUMENT CONTROL DATA

1 ORIGINATING ACTIVITY	2a REPORT SECURITY CLASSIFICATION
THE RAND CORPORATION	UNCLASSIFIED
	2b GROUP

3. REPORT TITLE
AN ON-LINE DEBUGGER FOR OS/360 ASSEMBLY LANGUAGE PROGRAMS

4. AUTHOR(S) (Last name, first name, initial)
Josephs, W. H.

5. REPORT DATE	6a. TOTAL No. OF PAGES	6b. No. OF REFS.
August 1969	28	---

7. CONTRACT OR GRANT No.	8. ORIGINATOR'S REPORT No.
DAHC15-67-C-0141	RM-6027-ARPA

9a. AVAILABILITY/LIMITATION NOTICES	9b. SPONSORING AGENCY
DDC-1	Advanced Research Projects Agency

10. ABSTRACT

A description of DYDE, an on-line debugging program for use by assembly-language programmers on third-generation IBM computers. The ability to debug a program on-line can mean a significant reduction in the programmer's debugging effort. Many such on-line systems are available for other computers, notably DDT and its derivatives for the PDP computers, but none is widely known for the 360 system operating under OS, where dumps and off-line traces must be used. DYDE (Dynamic Debugger) provides the programmer with DDT-like capabilities of break-point insertion, modification of data and code, and symbolic-label references. It can operate under any option of OS/360, using only a small, user-written "ping-pong" SVC and communicating with the programmer by means of the 1052 operator's console (employing WTO and WTOR) or, preferably, the 2260 graphics console (using the OS Graphics Access Method support).

11. KEY WORDS

Computer programming
Computer graphics
DYDE (On-line Debugging Program)