701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

Xerox FORTRAN Debug Package (FDP)

Sigma 5-9 Computers

Reference Manual

FIRST EDITION

90 16 77A

July 1970

Price: $2.50

Printed in U.S.A.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 5 Computer/Reference Manual	90 09 59
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 8 Computer/Reference Manual	90 17 49
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox Extended FORTRAN IV/LN Reference Manual	90 09 56
Xerox Extended FORTRAN IV/OPS Reference Manual	90 11 43
Xerox Extended FORTRAN IV-H/LN Reference Manual	90 09 66
Xerox Extended FORTRAN IV-H/OPS Reference Manual	90 11 44
Xerox Universal Time-Sharing (UTS)/TS Reference Manual	90 09 07
Xerox Batch Time-Sharing Monitor (BTM)/TS Reference Manual	90 15 77
Xerox Batch Processing Monitor (BPM)/BP, RT Reference Manual	90 09 54

Manual Content Codes: BP - batch processing, LN - language, OPS - operations, RBP - remote batch processing, RT - real time, SM - system management, TS - time-sharing, UT - utilities.

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their Xerox sales representative for details.

CONTENTS

APPENDIXES

ILLUSTRATIONS

TABLES

1. INTRODUCTION

The FORTRAN debug package (FDP) is designed to be used with XDS Extended FORTRAN IV or FORTRAN IV-H, and it operates under Sigma 7 Universal Time-Sharing System (UTS), Sigma 5/7 Batch Time-Sharing Monitor (BTM), and Sigma 5/7 Batch Processing Monitor (BPM). An addition to the FORTRAN run-time library, the debug package is made up of special library routines that are called by FORTRAN object programs compiled in the debug mode. These routines interact with the program to detect, diagnose, and often allow temporary repair of program errors.

Note: For debugging runs, the main program must be compiled in debug mode.

The debugger can be used in batch and on-line mode. An extensive set of debug commands are available in both cases. In addition to the debug commands, the debugger has a few automatic debug features. One of these features is the automatic comparison of standard calling and receiving sequence arguments for type compatibility. When applicable, the number of arguments in the standard calling sequence is checked for equality with the number of dummies in the receiving sequence. Calling and receiving arguments are also tested for protection conflicts. Another automatic feature is the testing of subprogram dummy storage attempts to determine if they would violate the protection of calling sequence arguments. (This feature is not available for FORTRAN IV-H programs.) These debug features are discussed in Chapter 5 and Appendix A.

Debug-mode compilation is not recommended for nondebug runs, because it produces larger and slower programs than nondebug-mode compilation.

Batch and On-Line Debugging Capabilities

While some debugging capabilities are reserved for on-line use and others for batch use, most debugging capabilities are available in both modes:

1. Capabilities Available in Batch and On-Line Mode
    a. Skipping FORTRAN statements
    b. Setting breakpoints
        1. Statement breakpoints
        2. Statement breakpoints on execution count
        3. Data store breakpoints
        4. Data store breakpoints when given values are attained
        5. Breakpoints on CALLs or function references (the values of arguments may be displayed or changed)
        6. Stops (breakpoints normally resume execution)
    c. Displaying data
    d. Changing the values of variables
    e. Branching
    f. Tracing flow
    g. Displaying flow history (or erasing flow history)
    h. Revoking debug commands
    i. Outputting debug display to a selected file (or to M:DO)
    j. Setting abort level (to abort only on certain classes of FORTRAN run-time errors)

    Note: Items c through i may be used immediately or as options to be exercised at specified breakpoints.

2. Capabilities Available Only in Batch Mode

   a. Displaying postmortem data

   b. Displaying postmortem flow history

3. Capabilities Available Only in On-Line Mode

   a. Interrupting execution

   b. Resuming execution

   c. Stepping

   d. Displaying flow history (backtracking)

   e. Restarting

   f. Rewinding program files

   g. Quitting the debugging run

Input/Output

Debugger input is entered via the source input file (M:SI). In batch runs, debug commands appear immediately before any data for the FORTRAN execution; the GO command must be the last debug command. In on-line runs, debug commands are read at the beginning of the run and also at execution stops. The prompt character @ is displayed to signal that a debug command is to be entered from the terminal keyboard.

Debugger output is placed in the diagnostic output file (M:DO) or, at the user's option, in a selected file (the DCB designation is F:UF). In on-line runs, it is necessary that M:DO always remain assigned to the terminal. Output consists of commanded displays, status messages, and error messages, and these may be intermixed with output from the FORTRAN program.

2. TYPICAL USE OF DEBUGGING COMMANDS

This chapter gives a preliminary description of the debugging commands. Detailed command descriptions are given in Chapter 5. The emphasis here is on typical use of the commands by the on-line user. On-line usage is stressed because the debugger is a more powerful tool for on-line debugging than for batch debugging. Appendix B illustrates examples of batch debugging.

The on-line user may issue commands any time the debugger displays the prompt character @ on the left margin of the printout page. This occurs at the beginning of the run and at each stopping point. Stops occur on aborts and normal execution stops, and they also result from certain debugging commands. (Stops are explained in more detail in Chapter 5.)

When the user responds to the prompt, he types one line at the terminal. The last character of that line must be one of the following characters: new line, carriage return, or line feed. These are all treated equivalently in debugger command processing and are represented in this manual by the character (RET). In the examples given below, this terminating character is not usually shown, but it has to be supplied so that the command line will be transmitted to the debugger.

The commands in this chapter are described in the following order:

GO	SKIP	GOTO
RESTART	AT	FLOW
REWIND	ON	NOFLOW
QUIT	ON CALL	HISTORY
ABORT LEVEL	ON CALLS	RESET HISTORY
step	PRINT	USE FILE
break	value change	USE ME
		KILL

GO Command

GO is used to start or continue execution.

RESTART Command

RESTART is used to redirect execution flow to the beginning of the main program. Before proceeding, the debugger prompts to allow the user to reinitialize data, rewind files, and issue other debugging commands. In the following example, the user resets the variable Z to zero, rewinds the file on unit number 64, and requests flow tracing before rerunning:

```
@RESTART
@Z = 0
@REWIND 64
@FLOW
@GO
```

REWIND Command

REWIND permits the user to position files at their starting point. Each file is designated by its unit number. A list of unit numbers can be used in order to rewind a series of files with one REWIND command; commas are used to separate the unit numbers. After rewinding the files, the debugger prompts again as in the following example:

```
@REWIND 64,65,66
@
```

QUIT Command

QUIT is used to terminate the debugging run. Control returns to the monitor as in normal execution termination such as CALL EXIT.

ABORT LEVEL Command

The debugger initially sets the abort level at its minimum (minimum abort level is 1) in order to gain control on any run-time error. By using the ABORT LEVEL command, the user can change that level. In the example below, the abort level is set at its maximum (15). As a result, any run-time error messages will be displayed, but the debugger will gain control and stop only on the most severe type of error.

```
@ABORT LEVEL = 15
@
```

The abort level is seldom changed by on-line users since they can continue or restart; however, batch users often raise the level to avoid aborting on the less severe run-time errors since the execution is terminated.

Step Command

The on-line user may "step" to the next executable FORTRAN statement by issuing only a terminating character (carriage return, new line, or line feed – shown below as (RET)). The current execution resumes until that next statement is reached. At that point the debugger stops, displays a position message, and prompts. Position messages are described in Appendix A; it is sufficient here to point out that at least the source line number of the statement will appear. In the following example, the user steps twice:

```
@(RET)
  15:
@(RET)
  16:
@
```

In this case, execution stops just before the statement having line number 16 (on the source listing).

The user may also step through any number of statements by supplying a count before the terminating character. Thus in the previous example, line number 16 can be reached by the following alternative procedure:

```
@2(RET)
  16:
@
```

Break Command

The on-line user may momentarily depress the BREAK key in order to force execution to stop at the next executable FORTRAN statement. This so-called "single break" results in the following type of message:

```
(BREAK key depressed once)
16: BRK
@
```

In this case the debugger stops just before the statement at line number 16 is executed.

If a single break does not stop execution soon enough, the user may depress the BREAK key again. This causes the debugger to discontinue the current execution, display the latest known position in a "double break" message, and

prompt with an @. The current execution cannot be resumed after a double break. To start the run again, the user must issue a RESTART or GOTO command, as in the following example:

```
(BREAK key depressed once)
(BREAK key depressed again)
DBL BRK AFTER SUB1/16: CAN'T GO OR STEP
@GOTO SUB1/16
  16:
@GO
```

In the above case, execution is discontinued after line number 16 in subroutine SUB1. Consulting his source listing, the user decides that it is reasonable to reexecute the statement at line number 16; so he issues the GOTO command. The debugger stops again, just prior to executing that statement, to allow the user to issue further commands. (In this case the GO command is issued.)

A single break is usually sufficient to stop execution, but it may take several seconds before the break message appears. Double breaks are recommended in only two circumstances – to interrupt high-volume output and to gain control when it is suspected that execution is looping.

SKIP Command

SKIP commands are used to bypass executable FORTRAN statements, preventing their execution. Suppose the program contains the following source line:

```
18:     IF (X<.001) CALL WRITER
```

During a debugging run the user may bypass this line with the following command:

```
@SKIP 18
@
```

Logical IF statements, such as the one at line number 18, have an interesting feature. They contain a substatement (in this case CALL WRITER). If the user only wants to bypass the substatement, he can use an offset (that is, +1) as in the following example:

```
@SKIP 18+1
@
```

A SKIP command can be used to bypass a series of statements. As an illustration, suppose the user has written the following main program:

```
1:      COMMENT -- MAIN DRIVER
2:      10          CALL SUB1
3:      20          CALL SUB2
4:      30          CALL SUB3
5:      40          CALL SUB4
6:                  STOP
7:                  END
```

For this debugging run the user only wants to check out the routine SUB4 and has not even loaded the other routines. When the run begins, the following command causes the unwanted statements to be bypassed:

```
@SKIP 2 TO 4
@
```

An equivalent bypass can be set up with the following command:

```
@SKIP 10S TO 30S
@
```

The above command references statement labels 10 and 30; the S immediately follows statement label numbers to notify the debugger that a statement label is referenced, not a line number. To illustrate the flexibility regarding such commands, the following are also equivalent to SKIP 2 TO 4 in the above program:

```
SKIP 2 TO 30S
SKIP 10S TO 4
SKIP 2 TO 2+2
SKIP 10S TO 10S+2
SKIP 30S-2 TO 30S
```

Many other combinations are possible.

Line numbers, statement labels, line numbers with offsets, and statement labels with offsets (such as +2 and -2 above) are known as "positions". Positions are also used in AT and GOTO commands, and they are sometimes useful in PRINT commands. Another type of statement label – the global label – is available in XDS FORTRAN IV but not in FORTRAN IV-H. Global labels are referenced in the same form used in the source program, for example, 99$. The $ immediately follows the label number.

Positions can almost always be preceded by a "qualifier" (except after "TO" in SKIP commands). A qualifier specifies which region of an overall program to use. To specify the main region, the qualifier is a slash (/), as in the following example:

```
@SKIP /2 TO 4
@
```

This example is equivalent to the earlier SKIP 2 TO 4 command.

There must be at least one blank between a command name (e.g., SKIP) and the main qualifier. To specify a non-main region, the qualifier consists of a FUNCTION, SUBROUTINE, or ENTRY name immediately followed by /, as in the following example:

```
@SKIP SUB4/19
@
```

Qualifiers may be used in front of variables and positions. It is always safe to use a qualifier, but it is not always necessary.

AT Command

An AT command is used to give the debugger control just prior to executing a given statement. In its simplest form the AT command resumes execution after notifying the user that the statement has been reached. For example,

```
@AT 19
@GO
/19:
  19:
  19:
```

In this case the user wants to be informed whenever the main program statement at line number 19 is encountered. He begins the run with the AT command and starts execution with the GO command. Each time that statement is reached, the debugger displays its position and resumes execution.

Suppose the user wants to be notified every third time that the statement is encountered. Then the following type of AT command can be used:

```
@AT 19 # 3
@
```

An AT command can be used to stop execution by supplying a STOP specification before or after the AT command:

```
@STOP AT 19
@GO
/19:
@
```

In the above example the statement at line number 19 is about to be executed, but the user now has the opportunity to issue further debugging commands. The next example stops the run just before the third execution (#3) of that statement (assuming that it had not been encountered prior to this time):

```
@AT 19 # 3 STOP
@GO
/19:
@
```

In this case if the user continues the run, it will stop again before the sixth execution of that statement. The whole sequence is repeated for illustration:

```
@AT 19 # 3 STOP
@GO
/19:              (third time)
@GO
 19:              (sixth time)
@
```

The AT command has another notable attribute – certain "attachable" commands can be attached to AT commands for added debugging capability. Some typical examples follow; note that semicolons are used to separate the commands. The following example requests flow tracing after reaching line number 28, and it discontinues the trace after reaching line number 39:

```
@AT 28; FLOW
@AT 39; NOFLOW
@
```

The example below requests that the value of X be displayed on reaching line number 50, and it directs the run back to line number 11 instead of executing the statement at 50:

```
@AT 50; PRINT X; GOTO 11
@
```

The next example is similar to the previous one, except that X is reset to zero after being displayed.

```
@AT 50; PRINT X; X = 0; GOTO 11
@
```

The AT command is not the only command allowing attachments. ON, ON CALL, and ON CALLS commands also permit attachments. The attachable commands are listed below for reference:

PRINT

OUTPUT

value change

GOTO

FLOW

NOFLOW

HISTORY

RESET HISTORY

USE FILE

USE ME

KILL

As indicated in previous examples, multiple attached commands are permitted. However, it is useless to attach anything to a GOTO command since execution resumes and any later attachments are ignored.

ON Command

The ON command is probably the most powerful debugging tool available to the user. ON commands assist in isolating bugs by informing the user when unexpected values are stored in certain variables. In the following example the user requests a trace of all stores into the variable I:

```
@ON I
@GO
/5:  I = 1
 5:  I = 2
 5:  I = 3
 5:  I = 4
 18:  I = 1
SUB5/7:  I = 0
/18:  I = 1
SUB5/7:  I = 0
/18:  I = 1
```

The user consults the source listing of his main program and finds the following:

```
 5:        DO 1, I = 1,3
18:        DO 2, I = 1,3
```

The trace shows that the first DO works properly but that the second DO is failing. According to the trace, the failure is caused by line number 7 in subroutine SUB5. The variable I is set back to zero at that point. Consulting the source listing of subroutine SUB5, the user finds the following:

```
 7:        INCHES = 0
```

Obviously, I and INCHES share the same location. The user therefore examines COMMON statements in SUB5 and in the main program and discovers an unintentional overlap.

The above example demonstrates that ON commands consider the location of a variable, not merely the name. In fact, they take into consideration all of the locations occupied by a variable. If any of these locations are stored into, the ON command takes effect. The key to this operation is that the debugger must be notified whenever a variable is stored into. Notification is provided by FORTRAN run-time library routines and by debug-mode FORTRAN programs. (However, "S in column 1" statements and nondebug-mode subprograms do not notify the debugger when they store into a variable.)

The variable used in an ON command may be a scalar (for example, I), an array element (for example, V(1) or M(2,1)), or a whole array (for example, V or M). When specifying an array element, the subscripts (or an element count) must be numeric.

ON commands allow attachments, and a STOP specification can be included before or after the command (see the example below).

ON commands can be made conditional on the variable attaining a given value. An example of a conditional ON command is

```
@STOP ON I = 0
@
```

(The STOP specification is not a requirement.)

The equal sign is only one type of relational operator that can be specified. It could have been any of the following:

```
.EQ.   or  =
.LT.   or  <
.LE.   or  <=  or  =<
.GT.   or  >
.GE.   or  >=  or  =>
.NE.   or  ><  or  <>
```

The value following a relational operator can be any constant recognized by the debugger that is consistent with the type of variable used. Some examples follow.

```
@ON V(1) > = .334 STOP
@ON COMPLEX  .EQ.  (-4,5)
@ON LOGICAL  .NE.  .TRUE.
@ON LOGICAL  .NE.  T
@ON LOGICAL >< T
@
```

The last three commands are equivalent.

The following example shows how a conditional ON command can be used to stop the run when any element of the vector V goes negative:

```
@STOP ON V < 0
@GO
SUBR/7:  V(3) = -.000125
@
```

In this case the element V(3) receives a negative value at line number 7 in the subroutine SUBR.

ON CALL Command

When a debug mode program is about to CALL or reference a (nonintrinsic) function, the debugger is notified. ON CALL commands take advantage of this operation. For example, suppose the user has defined a statement function named ASF, and he wants to be informed whenever it is used by his program:

```
@ON CALL ASF
@GO
/15:  CALL ASF
  16:  CALL ASF
  44:  CALL ASF
```

In this case, the function is used at line numbers 15, 16, and 44 of the main program.

An ON CALL command can contain attachments or a STOP specification. During processing of an ON CALL, the user may request that arguments of the call be displayed or changed. In the next example the attachment causes the first argument to be displayed, and the user then changes that value before continuing execution:

```
@STOP ON CALL SUB1; PRINT ARG.1
@GO
/19: CALL SUB1  ARG.1 = -1
@ARG.1  =  0
@GO
```

In this case SUB1 is called at line number 19 of the main program, which appears in the source listing as follows:

```
 19:          CALL SUB1 (IVAL,IPRIME)
```

When the user gives the command "ARG.1 = 0", IVAL is zeroed.

To change or display all the arguments, "ARGS." may be used in preference to listing each argument separately. An example is shown for the ON CALLS command, below.

ON CALLS Command

The ON CALLS command is a version of the ON CALL command that applies to all calls and functions. It is often used to provide a limited flow trace of the program. The following ON CALLS command will display each CALL and (nonintrinsic) function reference; furthermore, it will show the values of all the arguments at each such call:

```
@ON CALLS; PRINT ARGS.
@
```

PRINT Command

PRINT is used to display the values of variables or arguments (see the ON CALL and ON CALLS commands). (OUTPUT is equivalent to PRINT.) Two examples follow, the second of which shows how a list of variables can be displayed with a single PRINT command:

```
@PRINT V (1)
.125000
@PRINT I,J,K
30
40
50
@
```

PRINT commands can be attached to AT, ON, ON CALL, and ON CALLS commands as in the next example:

```
@AT 19; PRINT V(1)
@ON K STOP; PRINT I,J
@GO
/19:  V(1) = .125000
 23:  K = 50    I = 30
J = 40
@
```

There are a number of variations to the PRINT command. These are specified later in the detailed description of the command (Chapter 5).

Value Change Command

Value change commands allow the user to modify variables. This was illustrated earlier in RESTART, AT, and ON CALL command examples. It is possible to set a whole array with one value change command. For example, the following zeros all elements of the vector V:

```
@V = 0
@
```

The value given to the right of the equal sign must be a constant, and that constant must be consistent with the type of the variable being changed.

GOTO Command

GOTO (or GO TO) is an attachable command used for branching. An attached GOTO was used in an example of the AT command. See the break command for an example of a direct (nonattached) GOTO. Note that the user may branch to any statement known to the debugger; the statement does not have to be labeled.

FLOW Command

The FLOW command is used to obtain a trace of critical junctures in the path of program execution. Between those points, the flow is either sequential or controlled by DO or REPEAT loops. The following points are traced, with sample flow trace messages shown for each point:

1. CALL statements and (nonintrinsic) function references:

```
/51:  CALL SUB1
```

2. RETURN statements and returns from statement functions:

```
SUB1/17: RETURN
/52:
```

   (The second line shows where the program returned to.)

3. GOTO statements and attached GOTO commands:

```
53: GOTO
65 (40S):
```

   (The second line shows where the program branched to – in this example it is line number 65, which is labeled 40.)

4. Arithmetic IF statements:

    66: IF

    69(50S):

    (The second line shows where the program branched to.)

5. Substatements of logical IF statements when the logical expression is true:

    70: LOGL IF TRUE

NOFLOW Command

NOFLOW (or NO FLOW) simply turns off the FLOW command.

HISTORY Command

HISTORY is used to find out the path the program took in reaching its current position. Only critical points are shown, similar to the FLOW command. The debugger retains the latest 50 such points, each of which correspond to one of the message lines shown for the FLOW command. The on-line user can "backtrack" through this path by requesting HISTORY and "stepping". A short example follows:

```
@HISTORY
@(RET)
/52:
@(RET)
SUB1/17: RETURN
@(RET)
/51: CALL SUB1
@(RET)
NO MORE HIST.
@
```

In this example the user requests HISTORY and issues his first backtracking command ((RET)). The critical point displayed is line number 52 in the main program. The user backtracks again to find out that line 52 has been reached because of a RETURN at line number 17 of subroutine SUB1. He backtracks a third time, and the resultant display shows that the subroutine has been called at line number 51 of the main program. The final backtracking command shows that the debugger has no further record of critical points in program flow.

The user could have requested that several points be shown at once by using a count. This can be done either in a backtracking command (for example, 3(RET))or in the HISTORY command. The latter is illustrated below.

```
@HISTORY 99
/52:
SUB1/17: RETURN
/51:  CALL SUB1
NO MORE HIST.
@
```

Note that the information is still shown in reverse flow order. (In batch runs, it is shown in flow order.) It should be clearly understood that backtracking does not change the path of flow; it merely reports what the path was.

RESET HISTORY Command

RESET HISTORY erases the current record of critical flow points. Its main value is that it allows the user to avoid duplications in HISTORY displays.

USE FILE Command

If the on-line user wants to leave the terminal in a short time, he can issue a USE FILE command. During execution, debugger output is placed in a designated file for later examination. In essence, the run shifts from on-line to batch debugging. In the following example, debugger output goes to a file named CHARLIE:

```
@USE FILE CHARLIE
@GO
```

During execution stops, the debugger displays its output at the terminal. Also, run-time error messages appear at the terminal, but in addition they are placed in the file.

USE ME Command

USE ME allows the user to stop using a file for debugger output. It closes the file designated by an earlier USE FILE command and directs subsequent debugger output to the terminal. Following is an example of an attached USE ME command:

```
@USE FILE FLOWCK
@FLOW
@STOP ON CALL WRAPUP; NOFLOW; USE ME
@GO
/77:  CALL WRAPUP
@
```

In this case a flow trace has been placed in the file FLOWCK. That trace can be examined after concluding the debugging run if the user wishes. For the present, however, he may proceed to debug his WRAPUP routine in the normal, on-line manner.

KILL Command

The user can revoke any SKIP, AT, ON, ON CALL, or ON CALLS command that is being used by the debugger. To do this, the user issues a KILL command specifying the command to be revoked. In the following examples the commands are issued and then revoked immediately:

```
@SKIP 5
@KILL SKIP 5
@SKIP 5 TO 9
@KILL SKIP 5                (only the first position is needed)
@AT SUB1/18
@KILL AT SUB1/18
@STOP AT 5 # 3
@KILL AT 5                  (STOP and # 3 are not needed)
@STOP AT 5 # 3
@KILL STOP AT 5 # 3         (but they can be supplied)
@ON V(1)
@KILL ON V(1)
@STOP ON V < 0
@KILL ON V                  (STOP and < 0 are not needed)
@STOP ON CALL JUMP
@KILL ON CALL JUMP
@ON CALLS; PRINT ARGS.
@KILL ON CALLS              (revoking a command automatically revokes its attachments)
@
```

It is sometimes useful to have a command revoke itself, as shown below:

```
@ON X > 5; KILL ON X
@AT JUMP/5; KILL AT JUMP/5
@GO
/15: X = 6.66667
JUMP/5:
```

In this case the user obtains the desired information, but he avoids undesirable repetition of such information if the program iterates or loops.

The most convenient KILL command is shown below:

```
@KILL
@
```

A KILL that does not specify a particular command revokes all stored commands and their attachments. It also turns off flow tracing and resets the history record.

When an attached KILL command is used, the KILL attachment is automatically revoked; that is, KILLs are suicidal.

Sometimes the user may want to revoke or change the attachments to a particular command. This is accomplished by reissuing that command with the desired change. It automatically replaces the old one. For example,

```
@AT 19; PRINT I
@STOP AT 19; PRINT J
@
```

The first command no longer exists.

3. DEBUGGER INTERFACING

The debugger interfaces with the compiled FORTRAN program, the FORTRAN run-time library, and the host monitor (UTS, BTM, or BPM). The compiled program contains calls to the debugger and tables used by the debugger; the run-time library contains certain routines used by and/or using the debugger; and the host monitor supervises debugger operation in the normal manner for user programs.

Debug mode compilation differs from normal FORTRAN compilation in that (1) a debug table is produced and (2) special calls are inserted in the program, referencing debugging routines. The debug table and special calls are described below.

Debug Table

A debug table contains four principal elements:

1. A full name list for the compiled program.
2. A source line table.
3. A statement label table.
4. A list of entry point (region) names.

Name List

The full name list for the compiled program allows access to all the variables used by the program, but it does not allow access to compiler-generated temps or dummy variable names. (See the NAME LIST statement in the XDS Sigma 5/7 Extended FORTRAN IV Reference Manual, Publication No. 90 09 56.)

Source Line Table

The source line table contains entries that correspond to statements (i.e., source line numbers)[†] in the source listing. Specifically, the source line table contains an entry for each of the following:

1. Statement function definition statement.
2. FORMAT statement.
3. "S in column 1" statement or substatement (to indicate an assembly language instruction).
4. END statement.
5. Executable FORTRAN statement. (For a list of executable and nonexecutable statements, see Appendix B of the XDS Sigma 5/7 Extended FORTRAN IV Reference Manual.)
6. Executable FORTRAN substatement in compound statements.
7. Executable substatement contained in logical IF statements.
8. Default statement (generated by FORTRAN IV as described below – default statements are not generated by FORTRAN IV-H).

Each entry in the source line table contains two items: the location of the statement or substatement and the source line number from the source listing for the compilation. The locations for statements are characterized as follows. For a statement function or a FORMAT statement, the indicated location contains a bypass branch. For an "S in column 1" statement, the indicated location contains the instruction assembled for the statement. For the rest of the listed statements and substatements, the indicated location contains a special debugger call – the statement check-in described below in "Special Calls".

The source line number is the line number of the statement in the source listing. A substatement uses the line number of the overall statement; thus, the source line table will contain two entries with the same source line number for, say, a logical IF statement.

[†] See "Source Line Numbers" in Chapter 4.

Note also that the source line table has no entries for some source lines; specifically, there are no entries for comment lines, continuation lines, and nonexecutable FORTRAN statements (other than END, statement function definition, or FORMAT).

Default statements are generated in certain special situations; however, their use should be avoided because they may cause confusion during debugging. These statements are generated in the following cases:

1. A STOP statement is generated if a main program could sequentially flow into or branch to a SUBROUTINE, FUNCTION, or END statement. The source line number of the SUBROUTINE, FUNCTION, or END statement is used for the default STOP statement; a source line check-in is also generated.

2. A RETURN statement is generated if a subroutine or function could sequentially flow into or branch to a SUBROUTINE, FUNCTION, or END statement (similar to case 1 above).

3. A CONTINUE statement is generated if DO or REPEAT terminal labels are omitted as in the following example

```
DO 99, K = 1,10
OUTPUT (108), V(K)
END
```

   (This is similar to case 1 above.)

4. An END statement is generated if the user neglected to supply one. The last actual source line number is used for the default END statement, and a source-line check-in is also generated.

Statement Label Table

The debug table also includes a statement label table which contains each statement label along with the name of the program region containing that label. Thus, the user can designate which label he wants if the same label is used in several regions. However, if the same label appears more than once in a region, only the first such label may be referenced in debug commands. This is not unduly restrictive because the user can always reference source line numbers to get at statements that have duplicate labels.

Entry Point Names

The debug table also includes a list of entry point (region) names. An entry point name exists for each SUBROUTINE, FUNCTION, or ENTRY statement in the compiled program.

Special Calls

Of the special calls to debugging routines, five are important to the user: the initialization call, statement check-in calls, data check-in calls, calling sequence calls, and entry point calls.

Initialization Call

An initialization call is made by the main program. For this reason, main programs must be compiled in debug mode if the user expects to use the debugger.

Statement Check-In Calls

Statement check-in calls, or source line check-in calls, are made for executable FORTRAN statements and substatements, END statements, and default statements. These check-ins provide a great deal of control and are used to step, break, or skip statements. Note that statement check-in calls are not made for assembly code statements (e.g., an S in column 1), FORMAT statements, or statement function definitions.

Data Check-In Calls

Data check-in calls are used to determine at what point a program variable is affected.[†] Check-ins are made for each of the following:

1. Assignment statements.
2. ASSIGN statements.
3. Variable settings for DO or REPEAT statements.
4. The setting of multiple dummy counters.
5. Library routines that set arguments: DVCHK, OVERFL, SETEOF, EOFSET, SSWTCH, SLITET, BUFFERIN, BUFFEROU, DECODE, and ENCODE.
6. Library routines that store into lists or buffers: BUFFERIN, 9BINREAD, 9INPUTL, and 9IEDIT. (Thus, data check-ins may occur during data input. When a list of variables is stored, a data check-in occurs for each variable in the order that it appears.)

Note that data check-in calls are not made in "S in column 1" statements or in the debugger itself (see "Value Change Command" in Chapter 5).

Calling Sequence Calls

Calling sequence calls are made prior to standard calling sequences, to allow the user to examine or change the values of arguments before entering the called routine. Specifically, they are made for each CALL statement and for each reference to the following:

1. FUNCTION statements.
2. SUBROUTINE statements.
3. ENTRY statements.
4. Statement functions.
5. Nonintrinsic library functions (see "Library Subprograms" in the XDS Sigma 5/7 Extended FORTRAN Reference Manual).

Entry Point Calls

Entry point calls are made for each SUBROUTINE, FUNCTION, or ENTRY statement. These calls have two purposes: first, they register the entry point so that messages output by the debug routines will clearly annotate the position in the program, and second, they link debug tables when two or more debug-mode compilations are loaded as a single program. The user has no control over entry point calls but should be aware of their effect. In particular, debug routines know where an "external" debug table is located only when one of the external entry point calls is exercised (via a CALL statement or function reference). Once the debug tables have been linked, they stay linked and cannot be unlinked by any debugging command (such as restarting). However, the debug tables are unlinked by reloading.

Note: The user can force early linkage. For example, to link up the debug table for the external subprogram SUB the following steps could be used:

1. Issue the command STOP ON CALL SUB.
2. Branch to and execute a statement that calls SUB.
3. Issue a stepping command (⏎).

As a result of these steps, linkage is accomplished without actually executing the subprogram.

[†]This determination is made by storage location rather than by variable name. Thus, if X is equivalent to Y and the user asks for notification of storage into X, he will be notified when either X or Y is stored into.

4. DEBUGGER COMMAND LANGUAGE

This chapter covers general rules used to form debugger commands, special symbols used to describe debugger commands, and data elements commonly used in the commands. The actual debugger commands are described in Chapter 5.

General

Debugging commands are simple but readable, and artificial codes are avoided. In general, many of the normal rules of English grammar apply to debugger commands. Blanks are used to separate words and should not be embedded in the middle of command words or specifiers. Furthermore, words should not be run together. The command syntax minimizes the use of the shift key.

Each debugging command occupies a single line; continuations are not allowed. The line is limited to 72 characters, and a new line character is automatically inserted at the 73rd character position.

No name (or "identifier") in a command may contain more than eight characters. Otherwise, the usual FORTRAN conventions hold; for example, identifiers must be made up of letters and decimal digits, the first of which must be a letter. (See "Identifiers", Chapter 2, XDS Sigma 5/7 Extended FORTRAN IV Reference Manual for the usual FORTRAN conventions.)

Only a limited number of commands can be stored at any one time. If the user inputs one command too many, it is rejected and an error message is printed. Before the new command can be accepted on reissuance, one or more of the existing commands must be revoked.

Typographical Conventions Used In This Manual

Chapter 5 describes each debugging command and its specifications. The following conventions are used in explaining the format of the commands or giving examples:[†]

1. Lowercase items are used to indicate user-supplied data; that is, they must be replaced with actual names, added parameters, variables, etc.

2. Capitalized items must be typed exactly as they appear.

3. All other symbols (except for brackets, braces, and ellipses) are typed exactly as they appear.

4. Items enclosed in brackets [ ] are optional.

5. Items stacked inside braces { } indicate a choice must be made by the user.

6. Ellipses indicate repetition. For example,

   [x]...

   means that x is optional and more than one successive x is allowed.

7. The special symbol (RET) is used for carriage return, new line, or line feed. In on-line operation, commands are always terminated by (RET). In batch operation, the user may omit it, but blanks are required to "fill out" the remainder of the 72-character record (characters after the 72nd are ignored).

8. Blanks are used for delimiting words and specifications. This does not imply that only blanks delimit words. Semicolons, commas, relational operators, plus or minus signs, slashes, and (RET) also delimit words and specifications. (The horizontal tab character is treated as a blank when debugging commands are interpreted.)

[†]In examples and descriptions it is always assumed that debug-mode FORTRAN compilations are applicable.

Common Command Elements

The following elements are used quite often in debugger commands:

1. Variables, which denote scalars, whole arrays, or array elements.
2. Constants.
3. Arguments, which may be generally or individually specified.
4. Positions, which specify statements or substatements by referring to source line numbers or statement labels.
5. Qualifiers, which are used to specify regions within a program.

These commonly used command elements are described in more detail in the following paragraphs.

Note: In this section and in Chapter 5, the terms "break", "statement break", and "data break" are frequently used. They function as follows. A "break" results in stopping execution of the program and going to the debugger; a "statement break" results in going to the debugger before executing the statement; and a "data break" results in going to the debugger when storing into a variable.

Variable

The debugger recognizes three types of variables: array elements, whole arrays, and scalars. Each of these types is designated by a name, and the name must appear in an appropriate debug table name list. (Note: the name list only contains variable names; it does not contain dummy names or temp cell names.)

Array Element

There are two ways to reference an array – by subscripting and by element count. Subscripting can be used for vector and nonvector arrays, while element count can only be used for nonvector arrays. To reference an array element in subscript form, write the variable name, followed by positive or negative integers in parentheses. The number of integers in parentheses must match the number of dimensions in the array. In the following sample array element, Z is a three-dimensional array:

```
Z(3,1,1)
```

Note: In a FORTRAN program, implied DO loops or variable names can be used as subscripts, but the FORTRAN debugger recognizes only positive or negative integers as subscripts.

To reference an array element by element count, simply write the variable and follow it with a natural number – an unsigned integer greater than zero – in parentheses. Examples of referencing array elements by element count are

M(3) Refers to the third element of the (nonvector) array M.[†] Say, for example, that the array M has the elements 1, 3, 5, 7, illustrated as

```
1   5
3   7
```

The element M(3) would be the third element, or 5.

Z(6) Refers to the sixth element of the (nonvector) array Z.[†] Say, for example, that the array Z has the elements 1, 4, 8, 12, 16, 20, 24, 28, 32, illustrated as

```
1   12   24
4   16   28
8   20   32
```

The element Z(6) would be the sixth element, or 20.

Array elements in debug commands are located by the debugger and checked to see if the elements are within the range of the array. Individual subscripts are not range-checked.

[†]Vector arrays are always expressed in subscript form.

Whole Array

A whole array is referenced simply by writing the array name (without subscript or element count). This is roughly equivalent to writing a series of references to each element in the array.

Scalar

A scalar is the most common kind of variable. For example, A, B, and C are scalars in the following FORTRAN statement:

```
A = B + C
```

Constant

There are constants for each type of data. And of the many kinds of constants, the debugger recognizes the following:

1. Integer constants
2. Real constants
3. Double precision constants
4. Complex constants
5. Double complex constants
6. Logical constants
7. Text constants
8. Hexadecimal constants

The forms for the first five types (integer, real, double precision, complex, and double complex) are as specified for "Numeric Input Strings" in Chapter 6 of the XDS Sigma 5/7 Extended FORTRAN IV Reference Manual (Publication No. 90 09 56). However, the forms for the last three types (logical, text,† and hexadecimal) have been modified for the debugger to simplify and speed up interpretation. These modified forms of the text and hexadecimal constants are also used by Symbol, Meta-Symbol, and DELTA.

Logical constants must begin either with a T or F or with a decimal point immediately followed by T (for true) or F (for false). Subsequent characters for the constant are optional, but if used they must be letters or decimal points. Examples of logical constants are

```
T
F
.TRUE.
.FALSE.
```

Text constants are written in the form

```
's'
```

where s is a string of characters representing the desired text. (Two consecutive apostrophes may be used to represent a single apostrophe in the text string; thus, 'AB' 'C' results in the text string AB'C.) The number of characters in each text string must be consistent with the type of variable affected by the text constant:

No. of Characters	Type of Variable Affected
1 to 4	Integer, real, or logical
5 to 8	Double precision or complex
13 to 16	Double complex

†The "text" constants recognized by the FORTRAN debugger are modifications of the "literal" constants described in the Extended FORTRAN IV Reference Manual.

If less than the maximum number of characters is specified (that is, if 1 to 3, 5 to 7, or 13 to 15 characters are specified), the string is left-justified and blanks are added on the right to fill out the string. Thus, the string 'AB' actually produces four characters ('AB␣␣') for integer, real, or logical type; the string 'AB' would be incorrect for double precision, complex, or double complex type, because too few characters are given.

Hexadecimal constants have the form

X's'

where s is a string of hexadecimal digits. Similar to text constants, the number of digits in the string must be consistent with the type of variable affected by the hexadecimal constant:

No. of Digits	Type of Variable Affected
1 to 8	Integer, real, or logical
9 to 16	Double precision or complex
25 to 32	Double complex

If less than the maximum number of digits is specified (that is, if 1 to 7, 9 to 15, or 25 to 31 digits are specified), the hexadecimal digits are right-justified, with leading zeros. Thus, the hexadecimal constant X'123' actually produces an eight-digit string (X'00000123') for integer, real, or logical type; the string X'123' would be incorrect for double precision, complex, or double complex type, because too few characters are given.

Argument

An argument may be examined and changed only during calling sequence breaks (see the ON CALL and ON CALLS commands in Chapter 5). The forms are

ARGS. (specifies all arguments in a calling sequence)

ARG.n (specifies a particular argument in a calling sequence)

where n is a natural number. Note the period in each form: it prevents misinterpreting the argument as a variable name. Examples of arguments are

ARG.2 (designates the second argument in the calling sequence)

ARGS. (designates all arguments in the calling sequence)

Position

A position is used to reference a statement or substatement and can be any of the following: a source line number, a statement label, a source line number with offset, or a statement label with offset. Positions may be qualified, with one exception – after "TO" in a SKIP command.

Source Line Number

Source line numbers are simply natural numbers corresponding to one or more (in case of substatements) entries in a source line table. If a statement in a FORTRAN program is simple, there will be one source line table entry corresponding to that source line number; but if the statement has substatements, there will be several source line table entries corresponding to each such source line number. The source line number refers to the position of a statement in a FORTRAN program. For example, if the 13th statement in a FORTRAN program is IF (A .EQ. B) GOTO 99, the source line number would be 13 and the source line table would have two entries: one with the source line number 13, pointing to the IF substatement, and the other also with the source line number 13, but pointing to the GOTO substatement.

There is a difference in the interpretation of duplicated (for substatements) source line numbers, depending on the debug command. For AT, GOTO, OUTPUT, and PRINT commands, the first of the duplicates is understood to be the one intended. However, for a SKIP command specifying a source line number, the debugger assumes that the user means to skip the whole line (all the duplicates). This assumption is not made if the user specifies a statement label or if he uses an offset.

Examples of using source line numbers in debug commands are

SKIP 13 (causes the program to skip the entire 13th statement)

STOP AT 13 (causes the program to stop just before the 13th statement)

Statement Labels

There are two kinds of statement labels – global[†] and local – and they have the form

$$n\begin{Bmatrix}\$\\S\end{Bmatrix}$$

where

n is the statement label number.

$ identifies the label as a global label.

S identifies the label as a local label.

Note: No blanks can be embedded in either form.

XDS FORTRAN IV allows the use of the same local label in different parts of a program (as long as an END LABELS statement is given before each new use of the label). The debugger assumes that within a given program "region" (defined below in "Qualifiers") the first occurrence of a specified label is the one intended.

Each statement label corresponds to a single entry in the source line table; it is associated with a statement, but not any substatements that may follow. For example, with the following statement

9 K = 3; J = 2

where 9 is the label, the command

SKIP 9S

causes the program to skip only the K = 3 statement.

Offsets

For added flexibility, offsets may be attached to source line numbers or statement labels. They permit the user to refer to substatements. An offset follows the source line number or statement label and consists of a plus or minus sign followed by a natural number:

$$\begin{Bmatrix}+\\-\end{Bmatrix}\text{offset number}$$

where

\+ indicates a later entry in the source line table.

\- indicates an earlier entry in the source line table.

offset number is the number of entries in the source line table to skip over; for example, an offset number of +2 would indicate to the debugger to go forward two entries in the source line table.

To illustrate the use of offsets in debug commands, consider the following portion of a FORTRAN program:

Source Line No.	Label	Statement
95		X = Y
96	7	IF (A .EQ. B) X = X + 1

[†]Global labels may not be used in FORTRAN IV-H.

Using the AT and SKIP commands, the user could reference these statements as follows:

Statement and Action Desired	Possible Commands: With Statement Label	Possible Commands: With Source Line Number
Skip X = Y	SKIP 7S-1	SKIP 95 SKIP 96-1
At X = Y	AT 7S-1	AT 95 AT 96-1
Skip IF (A .EQ. B)	SKIP 7S	SKIP 95+1
At IF (A .EQ. B)	AT 7S	AT 96 AT 95+1
Skip X = X + 1	SKIP 7S+1	SKIP 96+1 SKIP 95+2
At X = X + 1	AT 7S+1	AT 96+1 AT 95+2
Skip IF (A .EQ. B) X = X + 1	- -	SKIP 96

Offsets are used to pass over entries in the source line table. But remember that the table may have more than one entry for some line numbers and no entries for other line numbers – more than one entry when a line contains a substatement and no entry when there is a continuation line or comment line. For example, look at the following portion of a FORTRAN program:

Source Line No.	Label	Statement
95	7	X = Y
96		IF (A . EQ. B) X = X + 1
97		T = 2
98		COMMENT--COMPUTE NEXT STEP
99		U = Z

The source line table for these statements will contain two entries for source line 96 (the first entry for the IF (A .EQ. B) substatement and the second for the X = X + 1 substatement) and no entry for line 98. Then, the command SKIP 7S+4 would reference the U = Z statement.

Qualifiers

Qualifiers are used to differentiate between various regions of a program. The main program defines a region and so does each SUBROUTINE, FUNCTION, and ENTRY statement. A qualifier in a debug command has the form

/ (to specify the main region; the character immediately before the slash cannot be a letter)

or

name/ (to specify a named region; the slash immediately follows the name – no embedded blanks)

where name is a FUNCTION, SUBROUTINE, or ENTRY identifier.

Qualifiers are important since so much duplication is allowed in a FORTRAN program. The same source line number, statement label, or variable name may be used in different subprograms; qualifiers are needed to identify the proper reference. To qualify an item, the user first writes the qualifier and then the item; for example,

SUB1/X (indicates that X is to be found in the debug table for subroutine SUB1)

/15 (indicates source line number 15 in the main program)

Qualifiers are not always necessary. Obviously if the program being debugged consists only of a main region, qualifiers are redundant. The following paragraphs describe the interpretation of qualified and unqualified items in debug commands.

Before presenting the qualification rules, it is necessary to understand how each of the three types of debug commands (direct, stored, and attachable)[†] is interpreted:

1. Direct commands are immediately interpreted and executed on input.

2. Stored commands are partially interpreted on input. If the command has a qualifier indicating a debug table and that debug table has been located previously by the debugger, then the command is "activated" and interpretation is completed (for the stored command, but not for any attachment). However, if the indicated table has not yet been located, the command is "deferred" until an entry point call check-in[††] occurs for the debug table. Once this call occurs, the command is interpreted and activated. Active stored commands are tested during the program run. When the requirements of the stored command are met, the command is exercised. The first time a stored command is exercised is critical if there are any commands attached.

3. Attached commands are only checked for syntax on input. Interpretation takes place when the accompanying stored command is first exercised. Thereafter, attached commands are executed without reinterpretation.

When debug commands are input or exercised, execution of the FORTRAN program will stop at a point called the "current point". The current point is determined by the latest check-in call, and it specifies the current debug table and the current program region. The current debug table and the main debug table are used in the following qualification rules:[†††]

1. If an item is qualified, only the indicated debug table is searched for that item.

2. If a stored command or direct command is <u>not</u> qualified, then the current debug table is searched when the command is input. If that search fails, the main debug table is searched (unless the current table is the main table). If both searches are unsuccessful, the command is rejected.

3. If an attachment to a stored command is <u>not</u> qualified, then the current debug table is searched when the command is first exercised. If that search fails, the main debug table is searched (unless the current table is the main table). If both searches are unsuccessful, the entire command is rejected.

Qualifiers may be used only in front of variable names, source line numbers, or statement labels. When used with variable names and source line numbers, the qualifier indicates a specific debug table; but when used with statement labels, the qualifier also indicates a specific region. This is important because of the strategy used by the debugger in hunting for labels.

Once a region is determined (either by qualifier or by default), that region is searched. If the search fails, the debugger searches the next region only if that region begins with an ENTRY statement. As long as there are no duplicate labels, statement labels can be referenced by using subroutine or function name qualifiers. A duplicate label can be referenced only if it is the first such label after an ENTRY statement; in that case, the qualifier to use is the entry name.

[†] These commands are discussed in more detail in Chapter 5.

[††] See "Entry Point Calls" in Chapter 3.

[†††] Of course the current debug table may actually be the main debug table; this happens by default during initial input of commands.

5. DESCRIPTION OF COMMANDS

Chapter 2 covered the typical use of each debug command, and Chapter 4 described notation conventions used in explaining debug commands and defined the following important debug command elements:

variables
constants
arguments
positions
qualifiers

This chapter specifies the syntax and operation of each debug command and describes execution stops and error-detection features.

There are three categories of debug commands – stored, attachable, and direct:

1. Stored commands are retained by the debugger and tested during program execution. Each time the requirements of a stored command are met by the program, the stored command is exercised.

2. Attachable commands may be attached to certain stored commands. When a stored command is exercised, any commands that are attached to it are also exercised. Attachable commands may also be used directly (that is, not attached to a stored command); usually this means they are executed immediately and are not retained. (Exceptions to this are postmortem HISTORY and postmortem PRINT commands.)

3. Direct commands are executed immediately and are not retained by the debugger.

Stored Commands

The following debug commands are stored commands:

SKIP
AT
ON
ON CALL
ON CALLS

These commands are retained by the debugger until replaced by a similar command or revoked by a KILL command. A stored command contains "unique identification", which is defined to be the command's required (not optional) parts. The following list shows only the unique identification of each stored command:

SKIP position
AT position
ON variable
ON CALL name (where "name" is a function or subroutine name)
ON CALLS

If two AT commands, for instance, are issued for the same position, the first command is automatically replaced by the second one.

Positions and variables are translated into corresponding memory locations. As a result, SKIP, AT, and ON commands are either "deferred" or "active". They are active when the corresponding memory location is known. Positions and variables in the main program can always be located. However, positions and variables in external subprograms are locatable only after the subprogram has been entered (see "Entry Point Calls" in Chapter 3). Until that time, such commands are deferred and do not participate in the debugging process.

The SKIP command has only one purpose – to prevent execution of FORTRAN statements. The remaining stored commands are multipurpose, depending on options selected by the user. By using a STOP specification, they can be made to halt execution. In on-line runs, this allows the user to obtain control and issue further debug commands. In batch runs, this produces display of any requested postmortem HISTORY or PRINTs before the run terminates.

Another multipurpose feature of AT, ON, ON CALL, and ON CALLS commands is that they may have attachments (that is, attachable commands added to them). This permits a stored command to perform a number of debugging functions.

STOPs and attachable commands are associated with a stored command in accordance with the following general form:†

[STOP] stored [STOP][;attachable] . . . ⏎

If more than one attachable command is supplied, they will be exercised one at a time in the order given after the stored command is exercised. However, a STOP takes effect only after the last attachable command has been exercised. (A STOP is ignored if a GOTO attachment is exercised.)

SKIP Command

SKIP is the only stored command that does not allow STOP specifications and attachments. SKIP commands may be used in on-line or batch runs. They prevent the execution of FORTRAN statements at selected positions. The general form of a SKIP command is

SKIP position [TO unqualified position] ⏎

Thus, a SKIP command may be used to skip a single statement or a series of statements. The positions in a SKIP command define the "range of the skip".

SKIP commands are exercised at the conclusion of a statement check-in call (see Chapter 3) when that statement is in the range of the skip. Execution then resumes at the next statement.

It is possible to branch into the range of a skip or to step to each statement in that range, and AT commands may be exercised within the range. FORTRAN statements in the range of a skip will be skipped in any case.

Positions in a SKIP command are interpreted in one of two ways. If a position uses a statement label or an offset, it is interpreted to mean only the indicated statement. However, if a position uses a line number (without an offset), it is interpreted to mean the whole line – including any substatements for the statement at that line number.

When the debugger receives a SKIP command, it attempts to translate the positions into corresponding memory locations. Translation is delayed, however, if the position is qualified by an entry point name for an external subprogram not yet entered during execution. In this case the SKIP command is deferred until the subprogram is entered.

The debugger rejects a SKIP command that would bypass the end of a subprogram (that is, an END statement). If a SKIP command bypasses the terminal statement of a DO loop or a REPEAT loop, a warning message is issued but the SKIP command is not rejected.

AT Command

AT is a stored command; attachments and STOP specifications are allowed. AT commands may be used in on-line or batch runs. They cause "statement breaks"; that is, they put the debugger in control before execution of a FORTRAN statement at a selected position. The general form of an AT command is

AT position [#n]

where n is an integer greater than zero. If the #n option is not used, a statement break occurs at each execution of the statement at the selected position. If the #n option is used, a statement break only occurs at every nth execution of that statement (every nth time <u>after</u> the AT command is issued).

An AT command is exercised during a statement check-in call for the selected position. When a statement break occurs, the debugger displays that position and exercises any attachments to the AT command. (See "Position Messages" in Appendix A.) If a STOP specification is used with the AT command, execution stops as described later in this chapter.

---

† "Stored" stands for any AT, ON, ON CALL, or ON CALLS command and "attachable" stands for any of the attachable commands specified later in this chapter.

When the debugger receives an AT command, it attempts to translate the position into a corresponding memory location. Translation is delayed, however, if the position is qualified by an entry point name for an external subprogram not yet entered during execution. In this case, the AT command is deferred until the subprogram is entered.

ON Command

ON is a stored command; attachments and STOP specifications are allowed. ON commands may be used in on-line or batch runs. They cause "data breaks"; that is, they put the debugger in control after data is stored in a selected variable. The general form of an ON command is

ON variable [relation constant]

The variable may be a scalar, array element, or a whole array.[t] The constant may be any constant recognized by the debugger, but it must be compatible with the type of the variable selected. The "relation" may be one of the following operators:

.EQ.	=	
.GT.	>	
.LT.	<	
.LE.	<=	=<
.GE.	>=	=>
.NE.	><	<>

If the operator is one of those listed in the first column, however, the first period in the operator must not immediately follow the name of the variable. A blank can be used between the variable and the operator.

There are two general kinds of ON commands – conditional and unconditional. An unconditional ON command has the form

ON variable

It causes a data break each time the debugger is notified that the variable has been stored into.

A conditional ON command has the form

ON variable relation constant

It causes a data break only when (1) the debugger is notified that the variable has been stored into and (2) the stored value satisfies the specified condition (that is, "value relation constant" is logically true).

An ON command is exercised during any data check-in call affecting the location (or locations) allocated for the selected variable. When a data break occurs, the debugger displays information and exercises any attachments to the ON command. If a STOP is used with the ON command, execution stops as described later in this chapter under "Execution Stops".

The information displayed on a data break includes the position of the statement[tt] causing the data break. (See "Position Messages" in Appendix A.) In addition, the variable is identified and its value displayed in the manner described for the PRINT command (see "Value Display" below).

When the debugger receives an ON command, it attempts to translate the variable into its allocated memory location. Translation is delayed, however, if the variable is qualified by an entry point name for an external subprogram not yet entered during execution. In this case the ON command is deferred until the subprogram is entered.

---

[t] If an ON command designates a whole array that is not a vector (for example, a matrix), the debugger displays the element count (not the subscripts) of the array element causing a data break. For instance, A(1) identifies the first element of the nonvector array A.

[tt] In the unusual case where an input/output list contains an expression making a nonintrinsic function reference, the position may indicate the function rather than the input/output statement. The HISTORY command can be used to find the position of the input/output statement.

ON CALL Command

On CALL is a stored command; STOP specifications and attachments are allowed – including attachments that display or change the value of arguments. ON CALL commands may be used in on-line or batch runs. They cause "calling sequence breaks"; that is, they put the debugger in control before entering subroutines or nonintrinsic functions. The general form of an ON CALL command is

ON CALL name

where name is the name used in CALLs or function references to the routine of interest.

An ON CALL command is exercised during calling sequence calls that specify the selected name. When a calling sequence break occurs, the debugger displays information and exercises any attachments to the command. If a STOP is used, execution stops as described later in this chapter under "Execution Stops".

The information displayed on a calling sequence break includes the position of the statement making that CALL or function reference. (See "Position Messages" in Appendix A.) In addition, the specified name is displayed following the word CALL.

ON CALLS Command

The ON CALLS command is a generalized version of the ON CALL command. It causes a calling sequence break for every CALL or nonintrinsic function reference. The form of this command is

ON CALLS

Note: In case the user wants to issue an ON command for a variable named CALLS, he must qualify the variable (for example, ON /CALLS).

The use of the ON CALLS command does not interfere with an ON CALL command. If a calling sequence break occurs for a name selected by an ON CALL command, that command is exercised. The ON CALLS command is not exercised in this case.

Attachable Commands

The following debug commands are attachable commands:

PRINT
OUTPUT
Value change
GOTO
FLOW
NOFLOW
HISTORY
RESET HISTORY
USE FILE
USE ME
KILL

Attachable commands may be "directly used" or they may be attached to certain stored commands. When directly used, attachable commands have the general form

attachable [;attachable] . . . (RET)

When attached to stored commands, attachable commands are issued in accordance with the following general form:[†]

[STOP] stored [STOP] ;attachable [;attachable] . . . (RET)

[†]See "Stored Commands" above.

Note that semicolons are used for separating the commands. A series of commands on a line are exercised in the order given – from left to right.

Directly used attachable commands are usually exercised immediately. In batch runs, however, HISTORY and PRINT (or OUTPUT) commands are retained since it would be pointless to display such information before executing the program. When directly used, these commands are exercised when the run stops (see "Postmortem HISTORY" and "Postmortem PRINT" below).

Attachable commands attached to a stored command are exercised each time the stored command is exercised. The first time an attachable command is exercised, its interpretation becomes final; that is, it does not change on later exercise of the command. This procedure mainly affects attachable commands (PRINT, OUTPUT, value change, and GOTO) that may select unqualified positions or unqualified variables. The resulting interpretation will correspond either to the current subprogram or to the main program. The current subprogram is the one containing the position displayed when the stored command is exercised. AT commands are always exercised at the same position. However, an ON command, ON CALL command, or ON CALLS command may be exercised in a number of subprograms. The user must consider this possibility when selecting positions and variables for attached commands. A safe rule in such cases is to qualify positions and variables when in doubt.

PRINT Command and OUTPUT Command

PRINT and OUTPUT are synonymous debug command words. For simplicity, only PRINT is used in this document.

PRINT is an attachable command that may be used in on-line or batch runs. Many variations and options apply to PRINT commands; so its general form is necessarily complex. Its complexity requires that certain command elements be defined before the general format is defined.

A PRINT command can display the value of the following items: variables, arguments, and positions. The value of a position is the source line number of that position. Common reasons for requesting position displays are (1) to clarify the debugger's interpretation of a position that contains an offset and (2) to verify that the debugger recognizes a particular position. (Comment lines, continuation lines, and most nonexecutable FORTRAN statements are not recognized; also, the position may reference an external subprogram not yet available to the debugger.) Arguments may be displayed only during calling sequence breaks; see the ON CALL and ON CALLS commands above. The value of an argument is usually displayed in the same manner as the value of a variable. This is described below under "Value Display". Two special display forms are available – IN HEX or IN TEXT. IN HEX produces hexadecimal value while IN TEXT produces string value (that is, display of EBCDIC characters). Thus, a print "item" is defined to be one of the following:

position

argument [IN HEX / IN TEXT]

variable [IN HEX / IN TEXT]

The user may request that a single PRINT command display a list of items (commas are used to separate items in an item list). The general form of a PRINT command is

PRINT [IN HEX / IN TEXT] [item] [,item]...

Table 1 indicates the meaning of certain PRINT commands. This table illustrates the following points:

1. If the PRINT command does not include an item list, then all known variables are displayed. A variable is "known" if it appears in the name list of a debug table available to the debugger (see Chapter 3). A blank line precedes the displayed variables for each name list.
2. If no IN HEX or IN TEXT appears in a PRINT command, ordinary values are displayed.
3. If any item is followed by IN HEX or IN TEXT, then the displayed value of that item is in hexadecimal form or string form.
4. If IN HEX or IN TEXT follows PRINT, then all displayed values are in hexadecimal form or string form (except for any items that contain their own IN TEXT or IN HEX specifiers).

Table 1. PRINT Commands

Command	Meaning
PRINT	Displays the ordinary values† of all variables known to the debugger.
PRINT IN HEX	Displays the hexadecimal values of all known variables.
PRINT X	Displays the ordinary value† of X.
PRINT X,Y	Displays the ordinary values† of X and Y.
PRINT IN HEX X,Y	Displays the hexadecimal values of X and Y.
PRINT X, Y IN HEX	Displays the ordinary value† of X and the hexadecimal value of Y.
PRINT IN HEX X, Y IN TEXT, Z	Displays the hexadecimal values of X and Z and the string value of Y.

†Ordinary values are described under "Value Display" below.

Postmortem PRINT

During on-line runs, a direct PRINT command is exercised immediately. However, during batch runs a direct PRINT command is stored so it can be exercised when execution stops. It is called a "postmortem PRINT" command. Postmortem PRINTs are exercised whenever execution stops provided that the debugger has control, which is normally the case. However, if a halt occurs accompanied by a monitor error message, the debugger does not regain control. Note: Postmortem PRINT commands may have PRINT attachments, but no other attachable command is allowed.

Value Display

Values are displayed as a consequence of ON commands and PRINT commands. ON commands only display variables, using ordinary value. PRINT commands may display positions, arguments, or variables, and they may show hexadecimal value, string value, or ordinary value.

Each value display may include three parts: an identifier, an equal sign, and the value. The identifier and equal sign are not shown for direct (on-line) PRINT but in all other value displays, they are shown.

The identifier for a position is essentially the same position used in the PRINT command. Arguments are identified as "ARG.n", where n indicates the nth argument. The identifier of a variable depends on the kind of variable used in the PRINT or ON command. If the variable was qualified, the qualifier is displayed. The name of the variable is then shown. For scalar variables, no further identification is needed. For array elements, however, the name is followed by subscripts or an element count enclosed in parentheses. If the PRINT or ON command designated an array element (as opposed to the whole array), then the value display uses the same designation but with blanks and plus signs omitted. In case of whole-array designations, the identifier for the value display includes either the subscript (for a vector element) or the element count (for an element of a nonvector array). The debugger cannot display the individual subscripts of a nonvector array element, because in general the same element can be accessed with a variety of subscripts.

Usually the identifier, equal sign, and value appear on the same line. However, the value is shown indented on the next line if there is any danger of exceeding the right margin of the current line.

As stated previously, the value of a position is its source line number. For variables and arguments, the displayed value depends on "type" (for example, integer, real, logical, etc.).

String values occur when a PRINT item uses the IN TEXT specification. A string value is displayed as a certain number of EBCDIC characters enclosed in single quotation marks. Nonprinting characters are shown as blanks,

except for carriage returns, line feeds, or tab characters which are output without modification. The number of characters output depends on the type of the item displayed. Four characters are output if the type is integer, logical, or single precision real (also for arguments having no type). Eight characters are output for complex and double precision real items; 16 characters are output for double precision complex items.

Hexadecimal values occur when a PRINT item uses the IN HEX specifications. They are also produced in cases where ordinary values would be incorrect; this is discussed below. A hexadecimal value is displayed as a certain number of hexadecimal digits enclosed in single quotation marks with a capital X before the first quotation mark. Eight digits are output if the type of the displayed item is integer, logical, or single precision real (also for arguments having no type). Sixteen digits are output for complex and double precision real items; 32 digits are output for double precision complex items. The IN HEX specification is valuable because it produces the exact value of an item as used by the computer.

Ordinary values usually conform to the formats specified for the OUTPUT statement in the XDS Sigma 5/7 Extended FORTRAN IV Reference Manual. Exceptions occur in the following cases:

1. For integer items, the debugger tests the value for the possibility that it represents a known statement label. If it does, the integer value and the label are shown. The label is enclosed in parentheses.

2. For real numbers (including double precision and both parts of complex items), two tests are made. If the number represents a known statement label, the label is displayed in parentheses; the real value (or values) are not displayed (since they are not normalized floating-point numbers). If the real number (or numbers) are not normalized floating-point numbers and do not represent a known statement label, the item's hexadecimal value is displayed.

3. For arguments that have no type (PZE in the XDS Sigma 5/7 Extended FORTRAN IV Operations Manual), the value is tested for representing a known statement label. If it does, the label is displayed in parentheses. Otherwise, the debugger displays an asterisk (*), and the user must consult his source listing to determine what the argument was (for instance, it may be the name of an external subroutine).

It should be noted that if an argument is itself a dummy for a particular calling sequence, the true argument is displayed, not the dummy.

Value Change Command

Value change commands are attachable commands that may be used in on-line or batch runs. They are used to alter the contents of variables. The general form of a value change command is

$$\left\{\begin{matrix}\text{variable}\\ \text{argument}\end{matrix}\right\} = \text{constant}$$

Arguments can only be changed during calling sequence breaks; see the ON CALL and ON CALLS commands above. Changing an argument is identical to changing the variable (scalar or array element, not whole array) represented by the argument. The constant must conform to the type of the variable affected. When the "variable" in a value change command is a whole array, each element of the array receives the constant value. This is sometimes useful for initializing an array.

No display occurs when a value change command is exercised. Value change commands do not cause data breaks (see the ON command above).

GOTO Command

GOTO is an attachable command. It may be used in on-line or batch runs; however, GOTO commands may not be directly used in batch runs (they may only be attached to stored commands). GOTO commands alter the path of execution by branching to a selected position. The general form of a GOTO command is

GOTO position

Blanks are allowed between GO and TO.

It is useless to attach any commands to a GOTO command. Execution resumes, ignoring any further attached commands.

When GOTO is attached to a stored command, exercising the GOTO command produces the same flow tracing or history data as a GOTO statement. However, directly used GOTO commands produce a special history display, "GOTO CMD--position", where the position message indicates the first executable FORTRAN statement reached by the GOTO.

Directly used GOTO commands resume execution only momentarily. As soon as a statement check-in call occurs,[t] the run stops, a position message[tt] is displayed, a prompt character (@) is displayed, and the debugger awaits commands. This procedure allows the user to issue stepping commands after a direct GOTO.

When exercising a GOTO command that is attached to a stored command, execution immediately resumes at the selected position. Any STOP specification for the stored command is ignored.

FLOW and NOFLOW Commands

FLOW and NOFLOW are attachable commands. They may be used in on-line or batch runs. The FLOW command turns on the flow trace display mode; NOFLOW turns it off. The forms of these commands are

```
FLOW
NOFLOW
```

Blanks are allowed between NO and FLOW.

When these commands are exercised as attachments to stored commands, the words FLOW and NOFLOW are displayed in the flow trace display mode changes. The default setting of this mode is off.

While the flow trace display mode is on, the debugger displays messages at the following points:

CALL statements
Nonintrinsic function references
RETURN statements
Returns from statement functions
GOTO statements
GOTO commands that are attached to stored commands
Arithmetic IF statements
Substatements of logical IF statements that have a "true" logical expression

Between these points, execution is either sequential or else looping because of DO or REPEAT statements. (However nondebug-mode subprograms and "S in column 1" statements do not participate in the flow trace.)

The formats of the flow trace messages are as follows:

1. CALL statements and nonintrinsic function references:

   ```
   position CALL name
   ```

   where position indicates the statement containing the CALL or function reference. (See "Position Messages" in Appendix A.)

2. RETURN statements and returns from statement functions:

   ```
   position1 RETURN
   position2
   ```

   where position1 indicates the statement making the return, and position2 indicates the statement returned to.

---

[t] See Chapter 3.

[tt] See "Position Messages" in Appendix A.

3. GOTO statements and attached GOTO commands:

   position1 GOTO
   position2

   where position1 indicates the latest known statement before the branch, and position2 indicates the first statement check-in call after the branch.

4. Arithmetic IF statements:

   position1 IF
   position2

   where position1 and position2 are the same as for the GOTO statement above.

5. Substatements of logical IF statements that have a "true" logical expression:

   position LOGL IF TRUE

   where the position indicates the substatement of the logical IF statement.

HISTORY and RESET HISTORY Commands

HISTORY and RESET HISTORY are attachable commands. They may be used in on-line or batch runs.

During the course of execution, a history record is updated with flow trace transactions. Each transaction corresponds to one line of print as displayed for flow tracing (see the FLOW command and also see the direct GOTO command). The history record only holds the latest 50 transactions at any one time. The HISTORY command is used to display these transactions. RESET HISTORY simply erases the current transactions; this is useful to avoid duplication when repeatedly using HISTORY commands. The general forms of these commands are

HISTORY[n]
RESET HISTORY

where n indicates the number of transactions desired. Whenever a transaction is displayed, it contains a position message. (See "Position Messages" in Appendix A.) For clear documentation, this position message always includes the appropriate qualifier, even though this may be redundant.

On-line use of the HISTORY command differs from batch use. When an on-line HISTORY command is exercised, transactions are displayed in reverse order, starting with the latest recorded transaction. The HISTORY command begins by displaying the n most recent transactions (if any n is used) and then sets "backtracking" mode. In backtracking mode the user may continue to display transactions by issuing "backtracking" commands. When no further transactions remain, the message NO MORE HIST. is displayed. It is useless to attach commands to a direct HISTORY command during on-line runs, because they are ignored when backtracking begins.

When a batch HISTORY command is exercised, the most recent n transactions are displayed in normal flow order, not backwards (if no n is used, the entire history record is displayed by default). If this produces all available transactions, the display ends with the message NO MORE HIST. However, if there are more than n available transactions, the display ends with the message DONE.

If debugger output is going to a file (see the USE FILE command below) during HISTORY display, this output has the appearance of a batch HISTORY command.

When HISTORY and RESET HISTORY commands are exercised as attachments to a stored command, the messages HISTORY and RESET HISTORY are displayed.

Postmortem HISTORY

During on-line runs, a direct HISTORY command is immediately exercised. During batch runs, however, a direct HISTORY command is stored so it can be exercised when execution stops. It is called the "postmortem HISTORY" command (there should only be one such command). Postmortem HISTORY is exercised whenever

execution stops – provided that the debugger has control. Normally, this condition is met; however, if a halt occurs accompanied by a monitor error message, the debugger does not regain control.

USE FILE and USE ME Commands

USE FILE and USE ME are attachable commands. They may be used in batch runs, but they are intended primarily for on-line use. The USE FILE command designates that debugger output be placed in a specified file. USE ME designates that debugger output is again to be displayed at the terminal (M:DO). The general forms of the commands are

```
USE FILE filename
USE ME
```

where filename consists of from one to eight characters, and ends on (but does not include) the first blank, semicolon, or (RET) encountered after those characters.

An open use-file is closed in the following situations:

Exercise of a USE ME command
Exercise of a QUIT command
Exercise of a GO or stepping command after display of the RDY TO STOP or RDY TO ABORT message
Exercise of a USE FILE command having a different filename from the current open use-file

When a use-file is opened, it is essentially rewound. Thus, the file cannot be added to once it has been closed during the run. The DCB designation for the use-file is F:UF.

During execution, debugger output is routed to the use-file (if open). Run-time error messages are placed in the file and are also displayed at the terminal. When execution stops, debugger output appears at the terminal only, and display continues at the terminal until the run resumes because of a GO or stepping command.

KILL Command

KILL is an attachable command. It may be used in on-line or batch runs. There are two types of KILL commands – general and specific. The general KILL command has the form

```
KILL
```

The general KILL command does the following:

Revokes all stored commands (including their attachments)
Turns flow trace display off (i.e., NOFLOW)
Erases the history record (i.e., RESET HISTORY)
Discontinues backtracking mode (see the HISTORY command)

The specific KILL command has the form

```
KILL stored command
```

where stored command is any SKIP, AT, ON, ON CALL, or ON CALLS command with the proper unique identification. (See "Stored Commands" above.)

When a specific KILL command is exercised, the designated stored command is revoked along with any attachments to the stored command. Furthermore (even if the debugger is unable to find the stored command), the specific KILL command is also revoked. In other words, KILL commands are suicidal.

It is possible to attach a KILL command to a stored command so that, when exercised, the stored command is itself revoked. Of course, any later attachments will be ignored since they are revoked along with the stored command. However, if this stored command contains a STOP specification, the run will stop after the command is revoked.

When killing a command that has a given identifier, the user should use the same identifier in the KILL command. There are cases when this is not necessary. For example, an AT or SKIP command might be identified by a statement label while the KILL might identify the corresponding source line number. Also, an ON command might be

identified by a subscripted array element while the KILL might identify the array element using its element count. This may only be done with active commands. Alternate forms cannot be used in deleting deferred commands.

Notification is provided when a "missed kill" occurs; that is, when the identified command is not found by the debugger. This is not a serious error because of circumstances similar to the following case:

```
SKIP 21 TO 25 (RET)
ON X = > 0; KILL SKIP 21 (RET)
ON D = > 0; KILL SKIP 21 (RET)
```

In the above case, the user wants to bypass source line 21 through 25 until either X or D become positive. If X goes positive first, the SKIP command is killed. Later when D goes positive, there is no SKIP command to kill, and the debugger simply issues a warning that it missed the kill and continues execution.

Direct Commands

The following debug commands are direct commands:

```
Single and double break
GO
QUIT
RESTART
REWIND
ABORT LEVEL
Stepping and backtracking
```

Of these commands, only GO and ABORT LEVEL are permitted in batch runs. Direct commands are exercised immediately after being input. No command may be attached to a direct command. Although not recommended, it is possible to attach a direct command to directly used attachable commands. They have the general form

```
[attachable [;attachable] . . .;] direct (RET)
```

(None of the "attachable" commands should be a GOTO command or a HISTORY command if this is an on-line run.)

Single and Double Break Commands

Single and double breaks are direct commands. Attachments are impossible. The break command may only be used in on-line runs. They are used to interrupt program execution so that new debug commands can be issued.

The single break command is issued by depressing the BREAK key once. The debugger responds by preparing to stop at the next statement check-in call (Chapter 3) and then resumes execution. When that call takes place, the debugger displays a message, displays a prompt character (@), and awaits commands. The following message format is used:

```
position BRK
@
```

(See "Position Messages" in Appendix A.) There may be a significant time delay between depressing the BREAK key and receiving the break message. This is particularly evident when output is being displayed, since the statement check-in call will not occur until the current output has been transmitted.

A single break is usually sufficient to gain control at the terminal. But the user may issue a double break command if control is not obtained within a reasonable time. To issue a double break command, the user depresses the BREAK key again after issuing the single break command. The debugger responds by interrupting execution (it cannot be resumed), displaying a message, displaying a prompt character, and awaiting commands (GO and stepping commands will be rejected). The following message format is used:

```
DBL BRK AFTER position CAN'T GO OR STEP
@
```

where the position reflects the latest position known to the debugger and includes the appropriate qualifier, even though this may be redundant. To start the FORTRAN program running after a double break, the user must first issue a RESTART or a GOTO command; he may then use GO or stepping commands.

The BREAK key must be depressed for a minimum of one-fifth of a second; otherwise, a spurious character may be transmitted instead of the break character. There should be an appreciable delay between successive BREAK key depressions. If the BREAK key is depressed repeatedly in rapid succession, the debugger may receive a single break instead of a double break. It is also possible for the debugger to react to a double break as if two single break commands were issued. This happens if the user issues the second break after the statement check-in call occurs for the first break but before the BRK message has been transmitted to the terminal. If the BREAK key is depressed while inputting at the terminal, the current command line is automatically erased.

Since execution cannot be resumed after a double break command, double breaks should be issued only when circumstances demand that the run be interrupted. One such circumstance occurs when the user suspects that the program is looping in nondebug mode (or "S in column 1") code. If the program is looping elsewhere, a single break will stop the run. The most common circumstance for using a double break is to interrupt high-volume output.

GO Command

GO is a direct command. Attachments are not allowed. The GO command is used in on-line runs to start or to resume execution. In batch runs, there must be exactly one GO command, and it is the last debug command. The form of the GO command is

GO ⏎

QUIT Command

QUIT is a direct command. Attachments are not allowed. The QUIT command may only be used in on-line runs. It is used to speedily terminate the run. Control returns to the monitor after files are closed. The form of the QUIT command is

QUIT ⏎

RESTART Command

RESTART is a direct command. Attachments are not allowed. The RESTART command may only be used in on-line runs. It directs the debugger to prepare to rerun the FORTRAN program. The form of the RESTART command is

RESTART ⏎

When RESTART is exercised, the debugger performs as follows:

1. Prepares to resume execution at the beginning of the main program (just after the initialization call – see Chapter 3).
2. Discontinues stepping if a stepping command is in use.
3. Reestablishes the main debug table as being the current debug table.
4. Erases the history record (i.e., RESET HISTORY).
5. Discontinues backtracking mode if in use (see the HISTORY command).
6. Displays a prompt character (@).
7. Awaits further debug commands.

The RESTART command does not reinitialize program data, and it does not rewind any files.

REWIND Command

REWIND is a direct command. Attachments are not allowed. REWIND commands may only be used in on-line runs. They are used to rewind files used by the FORTRAN program (not the use-file). The general form of the REWIND command is

REWIND n[,n]. . . ⏎

where n is the unit number for the file to be rewound. Note that commas (not semicolons) are used to separate the unit numbers if more than one file is to be rewound by the command. Each unit number must satisfy the following inequality:

$1 \leq n \leq 65535$

ABORT LEVEL Command

ABORT LEVEL is a direct command. Attachments are not allowed. It may be used in on-line or batch runs. When run-time errors are detected, diagnostic information is displayed and execution either resumes or stops depending on the error severity level. (See Appendix A.) The error severity level is compared to the abort level, and if it is equal to or greater than the abort level, the run stops. The debugger assumes the minimum possible abort level (1), which stops execution on any run-time error. By using the ABORT LEVEL command, this can be changed. The general form of the ABORT LEVEL command is

ABORT LEVEL = n (RET)

where n is the desired abort level and is an integer satisfying the following inequality:

$1 \leq n \leq 15$

The abort level is seldom changed by on-line users, but batch users often raise the level to continue execution despite errors. (Nondebug-mode FORTRAN runs usually use an abort level of eight.)

Note: If the FORTRAN program specifies an alternate abort exit, this exit is taken and the run does not stop. (See, for instance, the ABORTSET routine in the FORTRAN run-time library.)

Stepping and Backtracking Commands

Stepping and backtracking are direct commands. Attachments are not allowed. They may only be used in on-line runs. The general form of these commands is

[n] (RET)

where n is an integer greater than zero. If the optional n is not used, the debugger assumes n = 1.

If the run is in "backtracking" mode (see the HISTORY command), then these commands cause backtracking; otherwise, they cause stepping.

Stepping

After any execution stop other than a double break, a stepping command causes execution to resume until n statement check-in calls (Chapter 3) occur. When the nth check-in occurs, the debugger displays its position (see "Position Messages" in Appendix A), displays a prompt character (@), and awaits further debug commands. A stepping command takes precedence over other commands. For instance, if the user steps to a statement designated in an AT command, the run stops before the AT command is exercised. If execution is resumed, the AT command may then be exercised – possibly resulting in another stop.

Backtracking

During an execution stop with backtracking mode set, the user may issue backtracking commands to display history transactions. In essence, he may step backwards through the history record. The intent of the backtracking command is as follows. The user requests backtracking mode with the HISTORY command. He backtracks to examine the n most recent transactions. After digesting this information, he may issue another backtracking command to look at the previous n transactions. This process continues until either the user has sufficient flow history information or no more information is available.

When all available transactions have been displayed, the debugger displays the message NO MORE HIST. and automatically leaves backtracking mode, prompting (@) for further debug commands. Backtracking mode is also terminated if the user issues one of the following commands: GO, GOTO, RESTART, or the general KILL command.

Backtracking commands do not erase the transactions they display. A second HISTORY command permits backtracking over the same transactions as its predecessor.

To obtain a complete backtrack of the history record, the user need only supply a large n ($n \geq 50$). This large n can be supplied in the HISTORY command as well as any backtracking command.

Error Detection Features

Errors are detected at three points during a debugging run:

By the debugger
By the monitor
By the standard FORTRAN run-time library

When errors are detected by the monitor, the debugger has no influence and does not regain control if the monitor aborts the run. When errors are detected by the standard FORTRAN run-time library, the debugger always precedes the FORTRAN run-time error message with a message of the following form:

position ERR

The position message indicates the latest statement known to the debugger when the error was detected. (See "Position Messages" in Appendix A.)

After a run-time error message has been displayed, the debugger is placed in control if two conditions are satisfied:

1. The error severity level is equal to or greater than the abort level.

2. No alternate abort exit has been specified by the FORTRAN program.

If the first condition is not satisfied, execution resumes at the point of the error. (This may cause later errors due to incorrect operation or data.) If the second condition is not satisfied, execution resumes at the statement specified as the abort exit; see the FORTRAN library routine ABORTSET and also see the "ERR=" specification for READ statements.[†]

However, if both conditions are satisfied, the debugger obtains control and takes action as follows:

- In on-line runs it displays the message RDY TO ABORT, displays a prompt character (@), and awaits commands. If the user issues a GO or stepping command, the run aborts. The message ABORTING is displayed and the use-file is closed before control returns to the monitor.

- In batch runs the debugger exercises any postmortem HISTORY or postmortem PRINT commands. Then the run aborts. The message ABORTING is displayed and the use-file is closed before control returns to the montior.

Error detection by the debugger covers four distinct classes:

Input/output errors while handling the use-file or debug command input

Possible errors in command usage (missed KILLs and skipping the terminal statement of a DO or REPEAT loop)

Command errors

Execution errors

The resulting error messages are fully described in Appendix A.

Command errors always result in rejection of the entire command line containing the error. The user is notified that the command is rejected and a diagnostic message indicates the nature of the error. Additional information is supplied if a command error is detected subsequent to storing the command. This may occur when activating a deferred command or when first exercising an attachment to a stored command. In on-line runs the debugger stops after detecting command errors. A prompt character is displayed, and the user has the opportunity to correct the problem. In batch runs the debugger resumes operation after rejecting the command containing the error; thus, the run may produce partial debugging results.

Execution error detection depends on the type of compilation used.

[†]Reference: XDS Sigma 5/7 Extended FORTRAN IV and FORTRAN IV-H Reference Manuals.

For XDS FORTRAN IV, the debugger performs the following checks. (This augments or replaces similar error-detection features used in nondebug-mode runs.)

1. It compares the number of arguments for a standard calling sequence to the number of dummies in its corresponding receiving sequence.†

2. It tests for type incompatibilities between arguments and dummies.†

3. It checks for "protection" mismatches between arguments and dummies.†

4. It tests for an attempt to store into a "protected" argument of a standard calling sequence.† This is illustrated by the following portions of a FORTRAN IV program:

```
main
program
          17:    CALL SUB (5.,X + Y)

subroutine
           1:    SUBROUTINE SUB (DMY1, DMY2)
           5:    DMY1 = DMY1 + 6.6
           9:    DMY2 = DMY1
```

The CALL statement at source line number 17 contains two protected arguments – the constant (5.) and the expression (X + Y). If the statement at line number 5 or 9 is executed, the debugger displays one of the following messages (before the argument is stored into):

```
SUB/5:   PROTECT ERR
SUB/9:   PROTECT ERR
```

The test for this type of error is made during data check-in calls (Chapter 3).

Note: XDS FORTRAN IV recognizes a "multiple dummy" used in passing a variable number of arguments. The multiple dummy has "type" and "protection" information like an ordinary dummy. This is compared against each argument that corresponds to the multiple dummy.

For XDS FORTRAN IV-H, the debugger performs more limited execution error checks. (For instance, no test is made for an attempt to store into a protected argument of a standard calling sequence.) Furthermore, these checks are made as a preliminary step. It is possible that the same error will be noted twice – first by the debugger and then by the standard FORTRAN run-time library. This approach ensures execution compatibility between debug-mode and nondebug-mode runs. The following execution error-detection checks are made for FORTRAN IV-H programs:

1. The debugger compares the number of arguments for a standard calling sequence to the number of dummies in its corresponding receiving sequence.

2. Except for statement functions, it tests for type incompatibilities between arguments and dummies.

3. Except for statement functions, it checks for protection mismatches between arguments and dummies.

Whenever the debugger detects one of the indicated execution errors, it displays an "execution error message"†† and assumes an error severity level of seven. If the abort level is greater than seven, execution resumes. If the abort level is less than or equal to seven, batch runs abort as described earlier. However, the on-line user is given the opportunity to issue debug commands. He may elect to resume execution (GO or stepping commands), but discretion is advised. If execution resumes after a run-time error, more errors may result because of incorrect operation or data.

† See the XDS Sigma 5/7 Extended FORTRAN IV Operations Manual

†† See Appendix A.

Execution Stops

During a debugging run, execution may stop for any of the reasons listed below:

1. The run stops after the initialization call (Chapter 3) so debug commands can be issued before the FORTRAN program starts (on-line only).

2. The run stops after any command has been issued other than a GO, stepping, or break command (on-line only).

3. The run stops after command rejection due to an error; thus, the user may correct the problem (on-line only).

4. The run stops when certain debug commands are exercised:

   a. Stepping or backtracking (see also the HISTORY command).

   b. Single break.

   c. Double break.

   d. Stored command (AT, ON, ON CALL, or ON CALLS) that uses a STOP specification.

5. The run stops on normal program stops (for instance, because of execution of a STOP statement or a CALL EXIT).

6. The run stops on FORTRAN run-time errors or debug-detected execution errors if two conditions are satisfied: (1) the error severity level is not less than the abort level (see the ABORT LEVEL command earlier in this chapter) and (2) no alternate abort exit is specified by the FORTRAN program. (An alternate abort exit is specified through use of the ABORTSET library routine. See also the "ERR=" specification for READ statements[†].)

7. The run stops if a PAUSE statement is executed.

8. The run stops if the monitor aborts the run or if a normal or abort exit to the monitor occurs.

If the stop is due to a monitor abort or exit (item 8 above), the debugger cannot regain control; the debugging run concludes.

When a PAUSE statement is executed (item 7 above), the message *PAUSE* is displayed. The user (in on-line runs) or the console operator (in batch runs) must respond with at least a (RET) character to resume execution. The debugger is not concerned with PAUSEs.

For the remaining stops (items 1 through 6 above), the debugger obtains control. In on-line runs it displays a prompt character (@) and awaits further debug commands. In batch runs it exercises any requested postmortem HISTORY or postmortem PRINT commands, closes the use-file (if open), and enters the standard FORTRAN library for exit to the monitor. If a normal (not abort) exit is to be taken, the library routine (9STOP) closes files used during the run.

[†]In XDS Sigma 5/7 Extended FORTRAN IV and FORTRAN IV-H Reference Manuals.

6. OPERATIONS

The debugger operates under the Universal Time-Sharing Monitor (UTS), the Batch Time-Sharing Monitor (BTM), and the Batch Processing Monitor (BPM). This chapter is mainly concerned with compiling, loading, and executing a FORTRAN program in debug mode in each of the three monitor systems.

Universal Time-Sharing Monitor (UTS)

FORTRAN IV programs may be debugged under UTS in either on-line or batch mode. (FORTRAN IV-H is not available under UTS.)

Batch Mode. For a description of how to submit a job in batch mode, see the XDS Sigma 5/7 Batch Processing Manual, Publication No. 90 09 54. Also see the discussion below under "Batch Processing Monitor" for a brief description of debugging a program in batch mode.

On-Line Mode. The operating procedures for on-line debugging under UTS are described briefly in the following paragraphs and in more detail in the XDS Sigma 7 Universal Time-Sharing (UTS) Reference Manual, Publication No. 90 09 07.[†] Figure 1 illustrates an on-line run of a simple FORTRAN IV program compiled, loaded, and executed in debug mode under UTS.

Logging On

On-line service is obtained from UTS by activating the user terminal and logging in. After the terminal is operational, the user alerts UTS by momentarily depressing the BREAK key. When the system is operative, the following messages will be printed:

UTS AT YOUR SERVICE[††]

ON AT time and data

LOGON PLEASE: user identification (RET)

!

An example of the proper log-in procedure is shown in Figure 1.

The exclamation mark in the last line of the above message informs the user that he is communicating with the Terminal Executive Language (TEL), the principal terminal language for UTS. Thereafter a prompt character is sent to the terminal following a completed request, an error, or an interruption by the user. If the services of another subsystem or processor are requested, the subsystem identifies itself with a different prompt character. The prompt characters used by TEL and subsystems that may be involved in debugging are as follows:

TEL	!
EDIT	*
FDP	@
FORT	>
Executing FORTRAN programs	?

The question mark (?) signals a request (READ, INPUT) for data to be input at the terminal.

[†] Hereafter referred to in this section as the UTS Reference Manual.

[††] Messages output by the terminal are shown underlined throughout this chapter, although in an actual session such Teletype output is not underlined. Characters without underscores are typed in by the user.

```
UTS AT YOUR SERVICE
ON AT 03:43 MAR 19,'75
LOGON PLEASE:   PAT,SANDY (RET)                    } Logging on

!EDIT (RET)                                        }
EDIT HERE                                          }
*BUILD SOURCE (RET)                                }
   1.000 *          READ(105,10) I (RET)           }
   2.000 *10        FORMAT(G) (RET)                } Building a source file
   3.000 *          STOP (RET)                     } file via Edit
   4.000 *          END (RET)                      }
   5.000 * (RET)                                   }
*END (RET)                                         }

!COMMENT ON ME (RET)                               Assigning DO (diagnostic output) to user's console

!SET M:SI DC/SOURCE (RET)                          Assigning SI to a file

!FORT4 (RET)                                       }
OPTIONS>  DEBUG (RET)                              } Compiling in debug mode
 HIGHEST ERROR SEVERITY:  0 (NO ERRORS)            }

!SET M:SI UC (RET)                                 Reassigning SI to the user's console

!LINK $ ON LMNFILE (PO) (RET)                      }
     LINKING $                                     } Linking the ROM output from the GO file

!START LMNFILE (RET)                               Initiating execution
@STOP AT 3 (RET)                                   User request for stop when source line 3 is reached
@GO (RET)                                          User request for program to start
?44 (RET)                                          Program request for input (caused by READ); user supplies 44
/3:                                                Stop produced by STOP AT 3 command above
@GO (RET)                                          User request for program to continue
*STOP* 0                                           Message produced by source line 3 (see SOURCE file above)
 3: RDY TO STOP                                    Message produced by debugger
@GO (RET)                                          User request for program to continue
STOPPING                                           Message produced by debugger

!OFF                                               }
                                                   } Logging off
CPU = .0323 CON= :06 INT = 22 CHG = .0000          }
```

Figure 1. Example of a Simple FORTRAN IV On-Line Program Run Under UTS

Compiling

The only difference between nondebug-mode and debug-mode compilation is that the debug option must be specified in debug-mode compilation.

In on-line UTS operations, the FORTRAN compiler is called by issuing the FORT4 command at the UTS executive level (TEL). This command specifies the files used by the compiler for input and output; its format is explained in the UTS Reference Manual. After the FORTRAN IV compiler has been entered in the on-line mode, it sends out a request for options. The user then specifies the debug option code (DEBUG) and any other options to be used in the compilation. (See the XDS Sigma 5/7 Extended FORTRAN IV Operations Manual, Publication No. 90 11 43, for the available options.) The Teletype printout during debug-mode compilation might appear as

```
!FORT4 ME (RET)

OPTIONS>  DEBUG (RET)
```

The first command calls the FORTRAN compiler and assigns the source input file to the terminal, and the second command designates debug-mode compilation. (Also see Figure 1 for examples of the procedure during debug-mode compilation.) If DEBUG is the only option specified at this time, the source input is read from the SI file, the relocatable object module (ROM) is output on the GO file, and the summary map is listed on the LO device.

After the options have been input, the compiler reads the source program from each of the files designated in the FORT command. Input continues until an END statement or end-of-file is encountered. Then the program summary and object program are output as requested and control is returned to the UTS executive (TEL).

The FORTRAN debugger reads input from SI. If SI has been assigned to a file earlier in the run, it has to be reassigned to the user's terminal before executing the program to be debugged. This should be done before the LINK command (see "Loading" below). In UTS, SI is reassigned to the terminal by the following command:

!SET M:SI UC (RET)

Not reassigning SI to the user's terminal can result in an end-of-file notice, an abort notice, and control being returned to the executive level of operation. If this happens, the user should reassign M:SI to the terminal and begin execution again.

Loading

In on-line UTS operations, the loading of programs is carried out by the LINK command. It takes the relocatable object modules produced by the compiler and links them together into a single executable program called a load module. To load and link a program compiled in debug mode, the user must use the P0 or FDP option with the LINK command; other than that, the format for the LINK command is the same as described in the UTS Reference Manual. The P0 or FDP reference "loads" the public library routines (P1) plus the debug routines. A simple example of the LINK command is

!LINK (FDP) (RET)

Also see the example of the LINK command in Figure 1.

When linking is completed, control returns to the UTS executive level.

Executing

On completion of the linking operation in on-line UTS, execution of a program compiled in debug mode is started by the START command. Or, if desired, both linking and execution can be initiated by a single RUN command. The forms for these commands are

!START [lm] (RET)
!RUN [mfl][,mfl]. . .[,mfl] { (P0) / (FDP) } (RET)

where

lm is the name (fid) of the load module to be executed; it should be the same load module that was used in a previous LINK command.

mfl is $ or the name (fid) of a relocatable object module to be linked and executed.

Note that FDP or P0 must be added to the RUN command, but not to the START command, to begin execution of programs compiled in debug mode. A reference to the debugger is not needed in the START command because of the previous FDP or P0 reference in the LINK command.

An example of the correct use of the START command is shown in Figure 1 and repeated here:

!LINK $ ON LMNFILE (FDP) (RET)
LINKING $
!START LMNFILE (RET)

With the RUN command substituted for both LINK and START commands, the example would appear as

```
!RUN $ ON LMNFILE (FDP) ⏎
    LINKING $
```

These commands are described in more detail in the UTS Reference Manual.

After the programs have been loaded into core, control passes to the debugger, which sends a prompt character (@) to the terminal and awaits commands.

Interrupting, Stopping, and Logging Off

When on-line execution is interrupted or stopped (see Chapter 5), the debugger prompts (with an @ symbol) for commands. At this time, the user may terminate debugger control by issuing a QUIT command which returns control to TEL. Control is also returned to TEL at the end of a debugging run.

An on-line session is ended by entering the OFF command at the UTS executive level:

```
!OFF ⏎
```

UTS then prints the accounting information (CPU time, terminal time, terminal interactions, and total charges), and the user terminal is automatically disconnected from the computer. An example of the logging off procedure is shown in Figure 1.

Gaining Access to the Use-File

During an on-line run, the user may want to assign debug output to a file and then examine that file later. The debug command USE FILE is used to assign debug output to a file (see Chapter 5). To gain access to the file in UTS, the PCL processor is used. The following sample printout illustrates the use of the COPY command which brings in the PCL processor:

```
!COPY DC/file TO ME ⏎
```

where file is the file name used with the USE FILE command, and ME causes the file to be printed at the terminal. After the file is printed, control automatically returns to the TEL level.

Batch Time-Sharing Monitor (BTM)

In BTM, FORTRAN IV-H programs may be debugged in either on-line or batch mode; FORTRAN IV programs are usually debugged in batch mode. (FORTRAN IV programs can be debugged on-line, but they must have first been compiled in batch mode.)

Batch Mode. For a description of how to submit a job in batch mode, see the XDS Sigma 5/7 Batch Processing Monitor Reference Manual, Publication No. 90 09 54. Also see the discussion below under "Batch Processing Monitor" for a brief description of debugging a program in batch mode.

On-Line Mode. The operating procedures for on-line debugging under BTM are described briefly in the following paragraphs and in more detail in the XDS Sigma 5/7 Batch Time-Sharing Monitor (BTM) Reference Manual, Publication No. 90 15 77.† Figure 2 illustrates an on-line run of a simple FORTRAN IV-H program compiled, loaded, and executed in debug mode under BTM.

Logging On

On-line service is obtained from BTM by activating the user terminal and logging in. When the system is operative, it will print the following messages:

```
BTM SYSTEM -x IS UP
date and time
!LOGIN: name,acct [,pass] ⏎
!
```

An example of the proper log-in procedure is shown in Figure 2.

†Hereafter referred to in this section as the BTM Reference Manual.

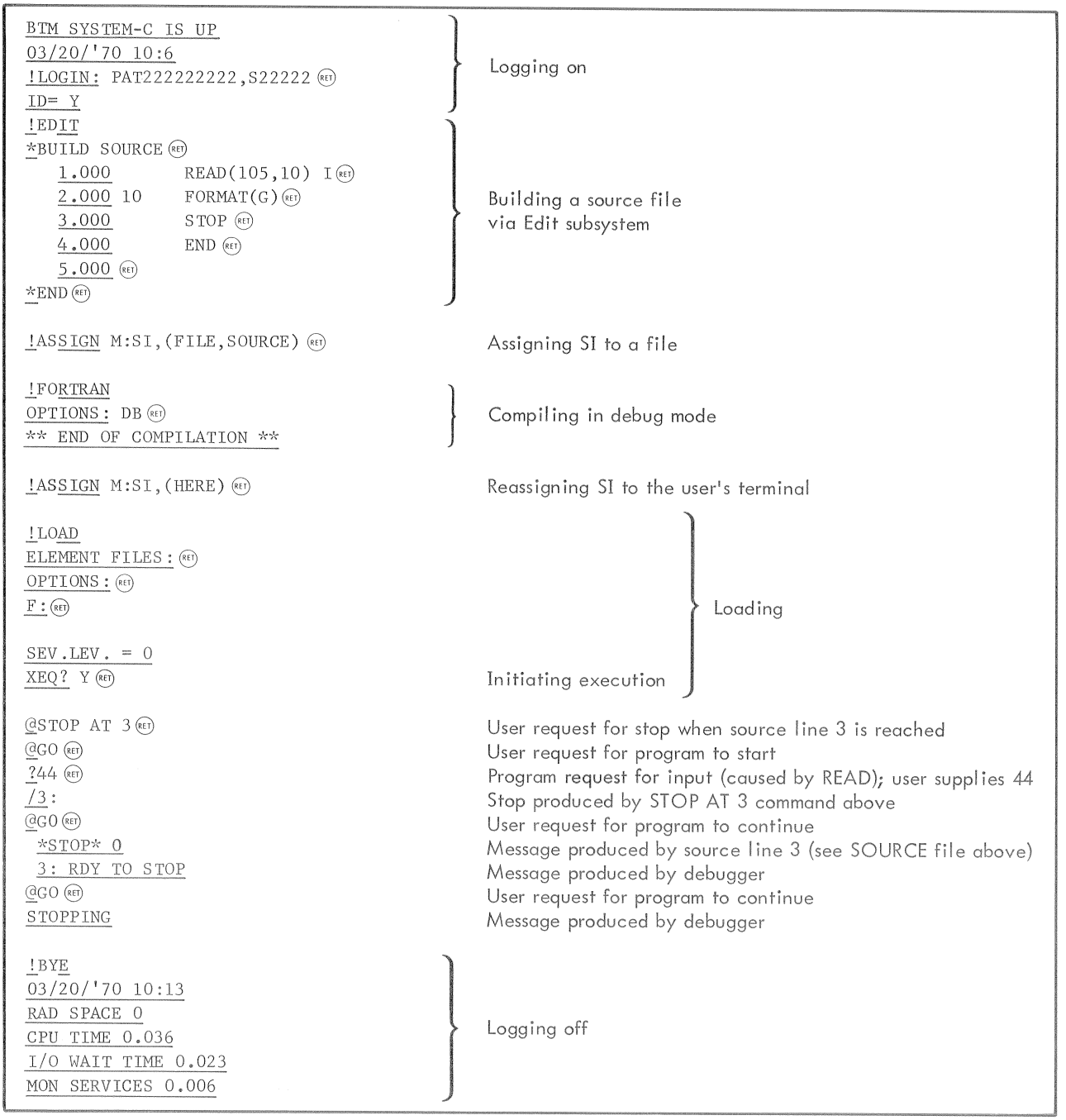


Figure 2. Example of a Simple FORTRAN IV-H Program Run Under BTM

The exclamation mark in the last line of the log-in messages informs the user that he is communicating with the BTM Executive. Thereafter a prompt character is sent to the terminal following a completed request, an error, or an interruption by the user. If the services of another subsystem or processor are requested, the subsystem identifies itself with a different prompt character. The prompt characters used by subsystems commonly involved in a debugging session are

BTM Executive	!
EDIT	*
FORTRAN compiler	>
Executing FORTRAN programs	?
FORTRAN debugger	@

The question mark (?) signals a request (READ, INPUT) for data to be input at the terminal.

Compiling

The only difference between nondebug-mode and debug-mode compilation is that the debug option must be specified in debug-mode compilation.

In on-line operations, the FORTRAN IV-H compiler is called by issuing the command FO at the BTM Executive level. (Files are either defaulted or are assigned by ASSIGN commands before the FO command.) After the FORTRAN IV-H compiler has been entered in the on-line mode, it sends out a request for options. The user then specifies the debug option code (DB for FORTRAN IV-H) and any other options to be used in the compilation. (See the XDS Sigma 5/7 Extended FORTRAN IV-H Operations Manual, Publication No. 90 11 44, for the available options.) The Teletype printout during debug-mode compilation might appear as

```
!FORTRAN
OPTIONS: DB (RET)
```

The first command calls the FORTRAN compiler, and the second command designates debug-mode compilation. If DB is the only option specified at this time, the source input is read from the SI file and the relocatable object module (ROM) is output on the BO file.

After the options have been input, the compiler reads the source program from the source input files. Input continues until an END statement or end-of-file is encountered. Then the program summary and object program are output as requested and control is returned to the BTM Executive.

The FORTRAN debugger reads input from SI. If SI has been assigned to a file earlier in the run, it has to be reassigned to the user's terminal before executing the program to be debugged. This should be done before loading the program. In BTM, SI is reassigned to the terminal by the following command:

```
!ASSIGN M:SI,(HERE) (RET)
```

Not reassigning SI to the user's terminal will result in end-of-file and abort notices and in control being returned to the executive level of operation. If this happens, the user should reassign M:SI to the terminal and begin execution again. An example of reassigning M:SI to the terminal is shown in Figure 2.

Loading

In BTM, programs compiled in debug mode are loaded in the same way as programs compiled in nondebug mode. Loading is carried out by the BTM Loader subsystem, which takes the relocatable object modules (ROMs) produced by the compiler and links them together for execution. The on-line user can enter the Loader subsystem by giving the command

```
!LOAD
```

When the Loader subsystem has been entered, it responds first with a request for the names of all element files ("ELEMENT FILES:" in Figure 2) from which the user wishes to load. If no element files are named, the Loader assumes default input from BO through M:BI. On accepting the element file list, the Loader issues a request for options ("OPTIONS:" in Figure 2). Here, the user responds with the options he wants. Then the Loader loads the specified ROMs and requests any DCBs ("F:" in Figure 2) that need to be specified. Figure 2 illustrates what happens during loading. For a detailed description of loading, see "Loader Subsystem" in the BTM Reference Manual.

Executing

Execution is actually started by the Loader subsystem. After programs have been loaded into core in on-line BTM, the Loader sends the message XEQ? to the terminal. If the user wants execution to begin, he types a Y and a carriage return. Control then passes to the debugger, which sends a prompt character (@) to the terminal and awaits commands. See Figure 2 for an example of initiating execution.

Interrupting, Stopping, and Logging Off

When on-line execution is interrupted or stopped (see Chapter 5) during debugging, the debugger prompts (with an @) for commands. At this time, the user may terminate debugger control by issuing a QUIT command which returns

control to the BTM Executive. Control is also returned to the BTM Executive at the end of a debugging run (that is, after the debugger message STOPPING or ABORTING).

An on-line session is terminated by giving the BYE command at the BTM Executive level:

```
!BYE
```

BTM then prints the accounting information (time-date, disc space, CPU time, I/O time, and monitor time), and the user terminal is automatically disconnected from the computer. An example of the logging off procedure is given in Figure 2.

Gaining Access to the Use-File

During an on-line run, the user may want to assign debug output to a file and then examine that file later. The debug command USE FILE is used to assign debug output to a file (see Chapter 5). To examine the file later in BTM, the FERRET and E[XAMINE] commands are used. The following sample printout illustrates the use of these commands.

```
!FERRET
>E[XAMINE] file ⏎
# ⏎
```

where file is the file name, and the last ⏎ causes the file to be printed at the terminal.

Batch Processing Monitor (BPM)

Both FORTRAN IV and FORTRAN IV-H programs can be debugged under BPM. Operating features for submitting a nondebug-mode batch job are described in the Sigma 5/7 Batch Processing Monitor Reference Manual, and sample deck setups for nondebug-mode batch jobs are shown in the FORTRAN operations manuals.[†] This section points out the features to be included for debug-mode batch processing. For purposes of illustration, sample debug-mode deck setups for FORTRAN IV and FORTRAN IV-H are included as Figures 3 and 4. Note that these decks differ from non-debug mode in the following ways:

1. A debug option must be specified on the FORTRAN processor control card: FORTRAN IV requires the option DEBUG; and FORTRAN IV-H, the option DB.
2. A "deck" of debug commands precedes the data for execution. A GO command must be the last card in the deck of debug commands (this is the only GO command that can be used in the debug deck).
3. Any data for execution follows the GO command.

Compiling

To specify debug-mode compilation in FORTRAN IV under BPM, the special option DEBUG must appear on the FORTRAN processor call card; for example,

```
!FORTRAN LS,GO,DEBUG
```

The DEBUG option may appear anywhere in the list of options. An example of this processor call card for FORTRAN IV is shown in Figure 3.

For FORTRAN IV-H, the option is DB instead of DEBUG. The same processor call card for FORTRAN IV-H would be

```
!FORTRANH LS,GO,DB
```

This card is shown in Figure 4.

[†]XDS Sigma 5/7 Extended FORTRAN IV Operations Manual (Publication No. 90 11 43) and XDS Sigma 5/7 Extended FORTRAN IV-H Operations Manual (Publication No. 90 11 44).

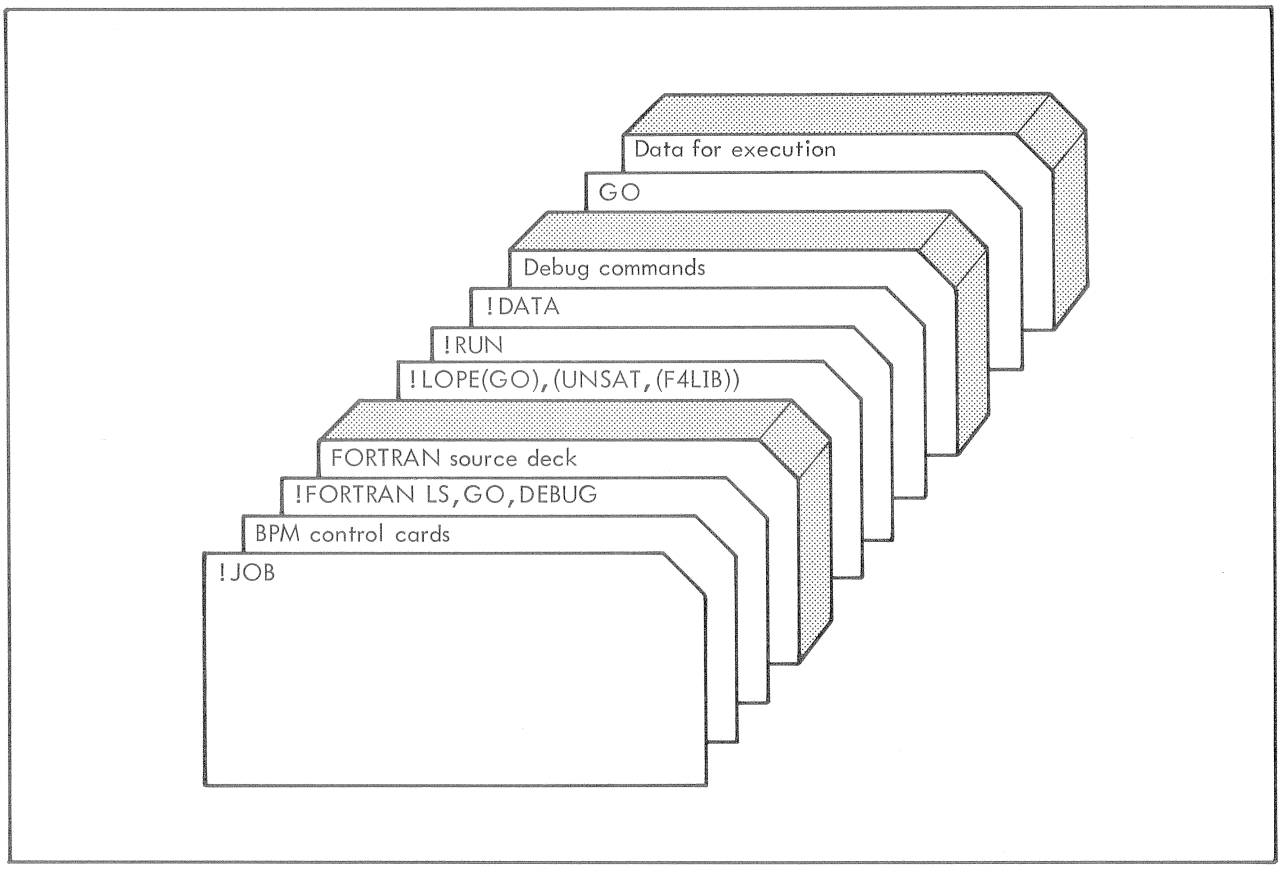


Figure 3. FORTRAN IV Deck Setup for Debug-Mode Batch Processing

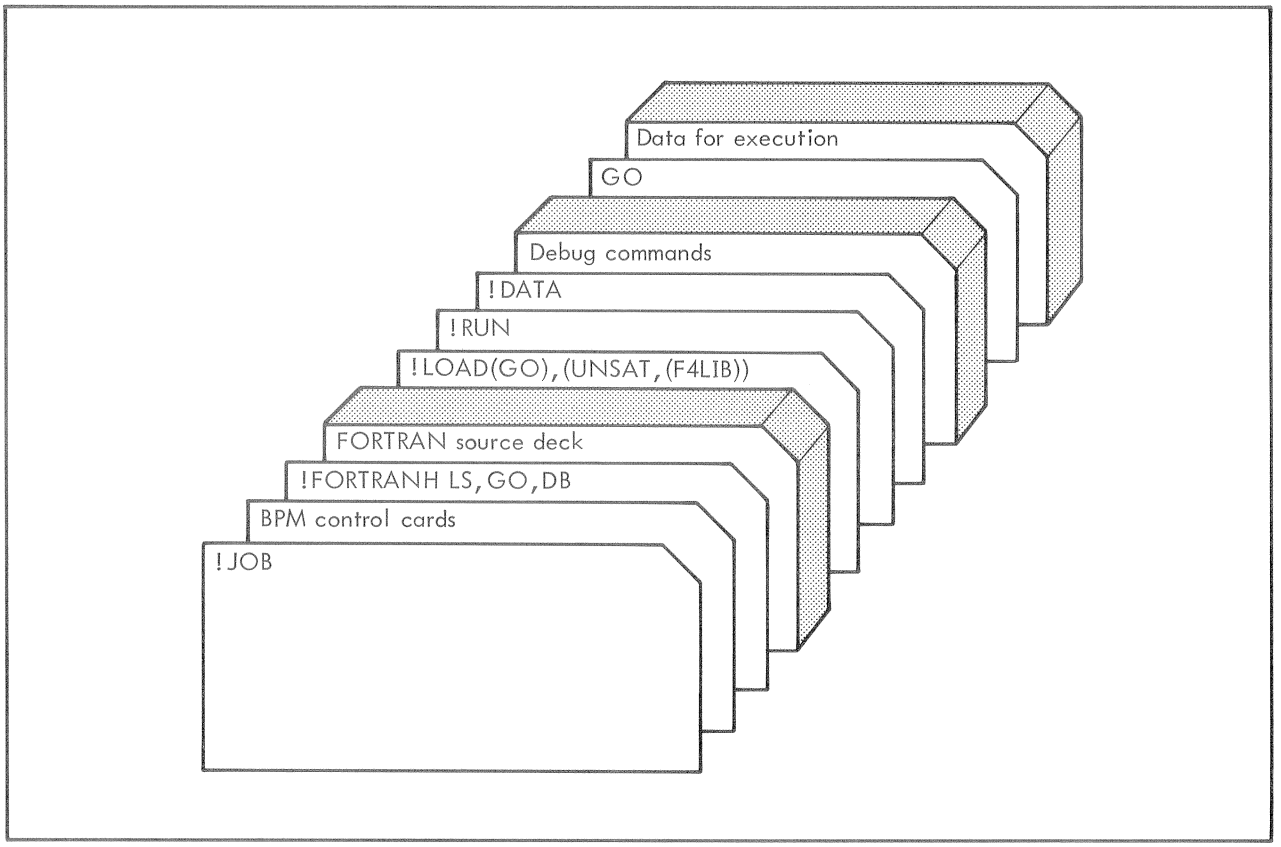


Figure 4. FORTRAN IV-H Deck Setup for Debug-Mode Batch Processing

Loading

In batch operations, programs to be debugged are loaded in the same way as nondebug-mode programs – namely, with a !LOAD card or a !LOPE card. To illustrate both commands, one has been included in Figure 3 and the other in Figure 4; actually, either command could have been used in each figure. For more information on these two load commands, see "Job Setup" in the Extended FORTRAN IV Operations Manual.

Executing

In batch operation, execution of debug and nondebug programs begins in the same way. However, debug-mode programs require that debugger input precede any data for execution. Debugger input consists of a "deck" of debugger commands,[†] terminated by a GO command. (It is this GO command that actually causes FORTRAN execution to begin.) Data for program execution immediately follows the GO command record. See Figures 3 and 4.

Use of FDP: UTS Versus BTM

Aside from different control languages, there are only two significant differences between UTS and BTM FORTRAN debugging: reaction to breaks and use of DELTA.

The reactions to single breaks may produce differing displays at the terminal. If the user hits the BREAK key while a message is being displayed, BTM may truncate it and redisplay the whole message again. UTS produces normal-looking messages despite single breaks.

The sophisticated user may want to use the machine language debugger, DELTA, in conjunction with FDP. This is more convenient under BTM than under UTS. Under BTM the BREAK key calls for FDP service; to call for DELTA service, the double ESCAPE key sequence (ESC ESC) is used.

If DELTA is used under UTS, it always takes control when the BREAK key is depressed. As a result, it is impossible to obtain the FDP single and double break command when DELTA is present. It is possible, however, to simulate an FDP double break command by using DELTA. The user need only hit the BREAK key to stop execution, thereby putting DELTA in control, and issue the DELTA command

```
location;G
```

where location is the proper point in the FORTRAN debug package. The location can be found in the following manner:

1. The user hits the BREAK key immediately after any FDP prompt character (@) is displayed. (It is recommended that this be done after the first such prompt.)

2. DELTA obtains control and issues a message containing a hexadecimal location, shown below as "xxxxx":

   ```
   BRK AT .xxxxx
   ```

3. In the current version of the FORTRAN debug package, the double break code is located 26 (hexadecimal) words beyond xxxxx. In fact, DELTA allows the use of the command ".xxxxx + .26;G" in place of "location;G".

4. After determining the needed location, the user may resume execution with the DELTA proceed command. When execution resumes, the prompt character (@) is redisplayed, and the user then issues FDP commands.

[†]Some examples of the use of batch commands are shown in Appendix B.

7. RESTRICTIONS AND LIMITATIONS

A number of restrictions apply to the FORTRAN debug package (FDP). Some are the result of restrictions in normal compilation and execution (for example, statement labels are restricted to the range 1-99999 and the abort level is restricted to the range 1-15). Other restrictions are peculiar to the use of FDP; these are discussed below.

Length of Command Input Line

The length of an FDP command input line is restricted to 72 characters. This allows character positions 73-80 to be used for sequence records.

Range of Source Line Numbers

Source line numbers are restricted to the range 1-10000.

Overlays

Overlays are not permitted in debugging runs. Since FDP must be able to search the chain of debug tables at any time during the run, the tables (and their associated subprograms) cannot be overlaid.

Not Available for Real-Time Runs

FDP is not designed to be used in a real-time run. The "real-time" FORTRAN run-time library differs from the "debug" FORTRAN run-time library. Furthermore, FDP must maintain interrupt control (for the BREAK key), and it does not save its own context when an interrupt occurs (that is, FDP is not self-interruptable or interrupt reentrant).

Nondebug-Mode Subprograms and Assembly Code

Nondebug-mode subprograms and assembly code (e.g., "S in column 1") are "invisible" to FDP. Their use can lead to gaps in flow or history traces and to overlooking critical data changes.

Output Constraints

The batch user should be selective in requesting output. If the M:DO file is filled, no further information is output; this may make the debugging run worthless and force a more selective rerun.

Length of Execution

Execution time is substantially increased because of the interaction between the program and FDP. Thus, debug-mode programs are not recommended for nondebugging runs (for example, production jobs). The amount of increase varies widely, depending on the FORTRAN program and on the FDP commands used. In some cases execution time may be doubled – particularly if several ON commands are used.

Program Size

The most severe limitation is program size. FDP alone occupies about 5,500 words of memory, and in addition, it requires certain run-time library routines (notably 9IEDIT) which occupy about 3,500 more words. Also, debug-mode FORTRAN programs are significantly larger than nondebug-mode programs because of the special calls and tables (especially the name list). The increase in size depends on the size of the FORTRAN program; generally it is less significant in big programs. Typical debug-mode programs tend to be 2.5 times the size of nondebug-mode programs.

The size increase is more noticeable when operating under BTM than under UTS or BPM, because the BTM user has a more limited portion of memory available to him. For instance, when the BTM user is limited to a 16K-18K user space, he can only load FORTRAN programs containing 100 to 200 average statements. (The loader also occupies some of the user space.) To overcome this limitation, the user may have to debug his program in several parts. A useful technique is to debug one subroutine at a time, using a special main program as a "driver" (a small program that calls the subroutine after making any needed setups). Note: Remember that both the main program and the subroutine have to be compiled in debug mode.

APPENDIX A. INFORMATION MESSAGES AND ERROR MESSAGES

Excluding messages produced by the user's FORTRAN program, there are three broad categories of messages:

1. Debugger messages.
2. FORTRAN run-time error messages.
3. Monitor error messages.

Debugger Messages

Debugger output can be grouped into the following general message classes:

Normal

Input/output error

Status

Execution error

Warning

Command error

Normal messages are described earlier in this manual in the descriptions of debug commands (Chapter 5). This appendix describes the remaining messages.

Input/Output Error Messages

Five input/output error messages can be produced by the debugger. These messages are described in Table 2.

Table 2. Debug Input/Output Error Messages

Message	Comments
CAN'T WORK USE-FILE	This rare message occurs if the debugger is unable to open or close the use-file. Processing continues as if a USE ME command had been given; that is, output is assigned to the terminal. The desired file is probably unavailable.
WRITE ERR, USE-FILE CLOSED	This rare message indicates that an error occurred while writing into an open use-file. The file is immediately closed, and the debugger proceeds as if a USE ME command had been given. The partial file may be available to the user when execution concludes.
CMD READ ERR, ABORTING	This message occurs in batch runs when an input/output error is detected while reading the debug commands from the source input file (M:SI). The run is aborted immediately.
CMD READ ERR, END-OF-FILE, ABORTING	During batch runs this error occurs if the source input file is empty. Note: There must at least be a GO command before batch debugging begins. During on-line runs this message occurs when the source input file (M:SI) has not been assigned to the terminal. In either case the run is aborted immediately; the on-line user can assign M:SI to the terminal and try again.

Table 2. Debug Input/Output Error Messages (cont.)

Message	Comments
CMD READ ERR, RETYPE	This message only occurs in on-line runs. Most often it results because M:SI is assigned to a file instead of the terminal. In such a case a series of these messages may appear. The user is advised to abort the job, assign M:SI to the terminal, and try again. If the message persists, he might try contacting the computer on a different communication line or changing terminals.

Status Messages

Ordinarily, debug-mode FORTRAN programs conclude execution with one of the status messages shown in Table 3. In this table the RDY TO STOP message (where RDY means ready) contains one of the debugger's most helpful displays – a position message.

Position Messages

A position message simply tells where an event has occurred as far as is known to the debugger. (If the debugger cannot determine the position, it uses the special message *UNKNOWN.) Position messages always contain a line number corresponding to one of the line numbers in a source listing of the program. If the position represents a labeled statement, the statement label is shown in parentheses after the line number. Position messages often begin with a qualifier, although this is omitted if it would be redundant; if the qualifier is omitted, the position is indented one space. A qualifier consists of a slash (/) for positions in the main program, and it consists of a function, subroutine, or entry name followed by a slash for positions outside the main region of the program. For message separation, positions are often followed by a colon. Some examples of positions are explained below.

Position	Explanation
/5:	Line number 5 in the main program.
SUBR1/5:	Line number 5 in SUBR1 (which is a subroutine, a function, or the name of an entry point).
/7(99$):	Line number 7 in the main program: the statement also has a global statement label 99$.
SUBR1/8(88S):	Line number 8 in SUBR1: the statement also has a local statement label 88.
9:	Line number 9 in the region of the program whose qualifier appears in a position message displayed earlier (the nearest such message).
10(105S):	Line number 10 in the same region: the statement also has a local statement label 105.

Execution Error Messages

There are five execution error messages (see Table 4). The first one, ERR, precedes FORTRAN run-time error messages. Its chief value is that it shows the latest position known to the debugger when the run-time error was detected.

The remaining four messages are produced during automatic checks by the debugger. These errors are similar to the FORTRAN run-time errors:

1. They have an associated error severity level (of 7).
2. If the abort level (see the ABORT LEVEL command) has been set higher than 7, execution resumes; however, this can be dangerous.

Table 3. Debug Status Messages

Message	Comments
position RDY TO STOP	This message results from execution of a normal stop (e.g., STOP statement or CALL EXIT statement). In batch runs this message is followed by any requested postmortem HISTORY or PRINTs and then the message STOPPING is given. In on-line runs the user may issue further debug commands; however, a step or GO command will end the debugging run with the message STOPPING.
STOPPING	This message indicates normal termination of the debugging run. The use-file is closed, if it is open, and control is transferred to the FORTRAN library routine 9STOP. That routine then closes any remaining open files and exits to the monitor.
RDY TO ABORT	This message only occurs in on-line runs. It signals that the run is about to be aborted because of a FORTRAN run-time error. The user may issue more debug commands; however, a step or GO command will end the run with the message ABORTING.
ABORTING	This message indicates an abort of the debugging run. In batch runs the message will be preceded by any requested postmortem HISTORY or PRINTs. The use-file is closed, if it is open, prior to the abort exit back to the monitor.

Table 4. Debug Execution Error Messages

Message	Comments
position ERR	This message precedes FORTRAN run-time error messages.
position ARG # ERR ON CALL name	This indicates that the number of arguments on the "named" CALL or function reference is not consistent with the number of dummies in the receiving routine. It is safe to resume execution if there are extra arguments (they will be ignored); otherwise, it may be dangerous to resume execution.
position type ARG.n VS type DUMMY ON CALL name	This shows a type incompatibility between the nth argument of the "named" CALL or function reference and its corresponding dummy. If execution is resumed, the wrong type of data will be used. An example of this message is 8: LOGL ARG. 1 VS INTG DUMMY ON CALL FUN
position PROTECT ERR, ARG.n ON CALL name	This indicates that the nth argument of the "named" CALL or function reference is protected (e.g., a constant), but the corresponding dummy signals that an attempt may be made to store into that argument. It is risky to resume execution.
position PROTECT ERR	This message is given if an attempt is about to be made to store into a protected argument (e.g., a constant). This is possible only when a subroutine or function has been called with one of the calling sequence arguments so protected. It is dangerous to resume execution; a constant may be destroyed, for instance.

3. If the abort level is equal to 7 or less (default value is 1), a test is made to determine if this is an on-line or batch run.

   a. In on-line runs the debugger sets to resume execution, but before resumption it requests that debug commands be given. It is dangerous to give a step or GO command in these situations; incorrect operation or data can result.

   b. In batch runs a test is made to see if the program specifies an alternate abort procedure (see the library routine ABORTSET). If so, execution resumes at that abort procedure. Usually, however, the run simply aborts; see the ABORTING status message in Table 3.

The automatic checks concern arguments and dummies. For a complete discussion of these topics, see the XDS Sigma 5/7 Extended FORTRAN IV Operations Manual. A few of the more pertinent points are covered here for convenience.

A CALL statement or (nonintrinsic) function reference generates a standard calling sequence. In debug mode this sequence contains the "name" used in the call. In addition, the number of arguments is indicated, and each argument has an associated "type" and protection bit – this bit signals if it is illegal to store into the argument (for instance, constants and expression temps are designated "protected").

When a standard call is performed, control passes to a subroutine, a function, an entry point, a run-time library routine, or a statement function defined in the FORTRAN program. These accept the standard call via a standard receiving sequence. The receiving sequence relates the arguments of the call to its dummies. The sequence indicates the number of dummies, and each dummy has an associated "type" and protection bit. For dummies the protection bit signals that the routine actually does store into the related argument. Thus, by matching the protection bits for a dummy and its argument, the debugger can detect attempts to store into a protected argument well in advance of the code that performs that store. Unfortunately not all receiving sequences contain the needed signal, but many of the run-time library routines do exhibit this signal, and the informed user can code receiving sequences (in assembly language) to meet this requirement. For added safety, however, debug-mode XDS FORTRAN IV compilations insert calls on the debugger prior to any code that actually stores via a dummy. Therefore, even if the receiving sequence test misses a protection mismatch, the debugger will catch the error before the store is attempted.

Finally, the debugger (via the standard receiving sequences) checks the arguments versus the dummies for "type" incompatibilities. The following type messages appear when an incompatibility is detected:

Type Message	Meaning
INTG	Integer
SNGL	Single precision real
DOUB	Double precision real
CMPX	Single precision complex
KMPX	Double precision complex
LOGL	Logical

Warning Messages

There are two warning messages (see Table 5). The debugger continues the execution in progress at the time that detection occurs, but the messages notify the user that an error may have developed.

Command Error Messages

When the debugger detects an erroneous command, it rejects the whole command containing the error. This has two notable ramifications. First, suppose a stored command is supplied that has a bad attachment. Then the entire stored command is rejected. Second, suppose a series of direct commands are strung out on a line as in the following example:

```
PRINT X; NFLOW; PRINT Y
```

If one of these commands is bad, then it and all remaining commands on that line are rejected. In the above example, this means that Y will not be displayed when the debugger encounters the bad NFLOW; however, X will be displayed.

Table 5. Debug Warning Messages

Message	Comments
MISSED KILL	This indicates that a KILL command has not taken effect because the debugger is unable to find the command that is supposed to be killed. The KILL command itself is then killed to prevent repetition of this message.
SKIPS LOOP END--qualifier (label)	This message warns that the terminal statement of a DO or REPEAT loop has been included in the range of a SKIP command. The "qualifier" and "label" conform to the description of positions given earlier in this appendix. It is usually improper to skip the terminal statement of a loop without also skipping the DO or REPEAT statement; however, the debugger makes no judgments about the user's intentions. An example of this message might be "SKIPS LOOP END--/(999S)". A statement labeled 999 in the main program (qualifier is /) is included in the range of a SKIP command, and 999 is the terminal statement label used in some DO or REPEAT statement. The SKIP command is accepted as it stands, however.

When composing command error messages, the debugger takes timing into account, supplying extra information in cases where the bad command may have been stored for awhile. This helps the user determine which command was rejected. Errors are detected at the following times:

1. Immediately after the command has been read (immediate errors).
2. Upon activation of a deferred command (activation errors).
3. At first exercise of an attachment to a stored command (exercise errors).

The resulting message formats are described below along with recommended recovery procedures for the on-line user.

Immediate Errors

If the debugger detects a bad command immediately after input, it gives an error message of the form

CMD REJECTED--message

The "message" indicates the nature of the error. To correct the error, the on-line user need only reissue the rejected command in its correct form.

Activation Errors

If the debugger finds errors while activating deferred commands, it gives one or more of the following forms of error messages:

qualifier DEFERRED-AT-CMD REJECTED--message

qualifier DEFERRED-ON-CMD REJECTED--message

qualifier DEFERRED-SKIP-CMD REJECTED--message

The "message" indicates the nature of the error, and the "qualifier" is the same qualifier used after the AT, ON, or SKIP command. An example follows. The user issues the command "ON SUBR2/W = 3; PRINT I" and starts or

continues execution. Assume that the debugger does not yet know about the external routine SUBR2; then it saves the command, deferring further action until SUBR2 is entered. Later, during execution, SUBR2 is entered, and the debugger begins activating that command but discovers that W is not a variable in SUBR2. The following message is given:

```
SUBR2/DEFERRED-ON-CMD REJECTED--CAN'T FIND W
```

To recover from activation errors, the on-line user need only reissue the rejected command (or commands) in corrected form.

Exercise Errors

If the debugger detects an error when first exercising an attachment to a stored command, it produces a two-line error message. The first line shows the type of attachment in error; one of the following lines will be displayed:

```
BAD GOTO
BAD KILL
BAD VALUE CHANGE
BAD PRINT
```

The second line shows the type of stored command containing that attachment as well as a "message" indicating the nature of the error:

```
stored command CMD REJECTED--message
```

For example, suppose the user issues the command "AT 12; GOTO 99" and starts or continues execution. After reaching line number 12, the debugger starts to exercise the GOTO 99 attachment but discovers that there is no line number 99 in its current or main program tables (perhaps 99 is a comment or continuation line). The following message is given:

```
BAD GOTO
AT-CMD REJECTED--CAN'T FIND 99
```

Recovery from exercise errors can be tricky. If the rejected command is an ON command, the user need only reissue a correct version. In other cases (SKIP, AT, ON CALL, and ON CALLS commands), reissuing the commands may not be sufficient. After supplying the corrected command, the user should give the original attachments as direct commands before proceeding; otherwise, the desired results will not be achieved until the next use of the (now corrected) command. In the example shown above, for instance, assume that the on-line user finds that 99 is a comment line and that he really wants to branch to line number 100. First, he issues the correct command as shown below:

```
BAD GOTO
AT-CMD REJECTED--CAN'T FIND 99
@AT 12; GOTO 100
```

As a result, the next time line number 12 is reached, the correct branch will occur. However, to obtain that branch at this time, the user must issue the GOTO directly.

```
@GOTO 100

100:
```

Having reached line number 100 via the direct GOTO command, the user may continue the run by using the GO command. (If he had given the GO command immediately after issuing the corrected AT command, the statement at line number 12 would have been executed – not the desired branch to 100.)

Table 6 shows the various "messages" that are included within the command error messages.

Table 6. Debug Command Error Messages

Message	Comments
CAN'T GO OR STEP	This message is given if the user tries to continue execution following a double break. To start execution again, the user first must issue a RESTART command or a GOTO command.
NO ROOM	This message indicates that the debugger is unable to accept any more stored commands. One or more KILL commands should be issued in order to make room for the new command.
MIS-USED	This message results if the following commands are attempted in a batch run: QUIT, REWIND, or RESTART. The message is also given if the following direct commands are attached to a stored command: a step command (number or new line following a semicolon), QUIT, REWIND, RESTART, GO, or ABORT LEVEL.
BAD CONST	This message indicates that an improper constant has been supplied in an ON command or a value change command.
BAD CHAR	This message indicates that a command contains an illegal character. (Usually a question mark is echoed at the terminal in place of the bad character.)
EH?	This message indicates bad syntax. Common causes are misspelling, missing separators (e.g., blanks), and using commas instead of semicolons. In batch runs, the question mark may be omitted after this message since the same line printers do not display that character.
SKIPS END	This message indicates that a SKIP command is rejected because it would skip the last statement — the END statement — in a program. (Otherwise, the executing run might "fall" beyond the FORTRAN code).
BAD SKIP SEQ	This message indicates that a SKIP command is rejected because the skip range is backwards or not sequenced properly. An example is "SKIP 9 TO 5".
ARG NOT ALLOWED	This message is given when an illegal ARG. or ARGS. reference is made. It is legal to print or change the value of an argument only after a calling sequence break; that is, after an ON CALL or ON CALLS command has been used.
NO ARG.n	This message indicates that there is no nth argument for a calling sequence break.
CAN'T CHANGE ARG.n	This message indicates that an argument value change command is rejected because that (nth) argument is protected. This occurs, for example, when the nth argument is a constant.

Table 6. Debug Command Error Messages (cont.)

Message	Comments
BAD TYPE ON ARG.n, USE type	This message indicates that a value change uses the wrong type of constant for the nth argument of a calling sequence break. For example, the message "BAD TYPE ON ARG.1, USE CMPX" indicates that the first argument is complex. See the description of the execution error messages earlier in this appendix for a discussion of type messages.
BAD TYPE ON name, USE type	This message indicates that a value change or ON command uses the wrong type of constant for the "named" variable. For example, the message "BAD TYPE ON L, USE LOGL" indicates that L is a logical type variable. See the description of the execution error messages earlier in this appendix for a discussion of type messages.
CAN'T FIND item	This message indicates that the debugger cannot locate the item in its appropriate table. The item may be a name, line number, label, or variable. In many cases, this message results from forgetting to use a qualifier.
BAD OFFSET = value	This message indicates that a position contained an offset that would go beyond program boundaries. The "value" is a plus or minus sign followed by the magnitude of the bad offset.
NON-ARRAY -- name	This message is given if a subscripted reference is made to the "named" variable, but that variable is not an array.
NOT IN ARRAY name	This message is given if a subscripted reference is not within the range of the "named" array.
BAD # SUBSCRS ON name	This message is given if too few or too many subscripts are supplied in a reference to the "named" array.

FORTRAN Run-Time Error Messages

Every message of this type contains the special identifying header

FORTRAN RUN-TIME ERROR

During debugging, this header is preceded by the execution error message

position ERR

The "position" indicates where the error occurred in the FORTRAN program (see the description of position messages earlier in this appendix under "Debugger Messages"). In some cases the FORTRAN RUN-TIME ERROR line also gives the name of the library routine applicable to that error plus a hexadecimal location value. (Since the debugger uses run-time routines for numeric input/output conversions, the special name *FDP* is used in case of conversion errors.) Following the header line, one or more lines are displayed to specify the exact nature of the error. If the severity of the error reaches the abort level (and assuming that no alternate abort procedure has been specified), then the debugger obtains control; see the RDY TO ABORT and ABORTING messages in the section mentioned above.

For a complete discussion of the FORTRAN run-time error messages, the reader may consult the XDS Sigma 5/7 Extended FORTRAN IV and Extended FORTRAN IV-H Operations Manuals. However, a few of the messages are listed below because their occurrence may mystify the FORTRAN user.

UNIMPLEMENTED INSTRUCTION TRAP AT X'loc'

PUSH DOWN STACK LIMIT TRAP AT X'loc'

DECIMAL ARITHMETIC TRAP AT X'loc'

WATCHDOG TIMER TRAP AT X'loc'

NONEXISTENT INSTRUCTION TRAP AT X'loc'

NONEXISTENT MEMORY ADDRESS TRAP AT X'loc'

PRIVILEGED INSTRUCTION TRAP AT X'loc'

MEMORY PROTECTION TRAP AT X'loc'

NON ALLOWED OPERATION TRAP AT X'loc'

The hexadecimal locations (loc) specify the computer location at which the problem was detected. Most of the above "trap" messages deal with improper use of machine instructions. Although normal FORTRAN statements use machine instructions properly, the programmer can obtain these errors when using "S in column 1" statements or subroutines written in assembly language. The programmer should take particular care not to modify register zero since serious errors may result.

If one of the above messages appear when using normal FORTRAN statements, the following list of error situations should be consulted:

1. Subscripts may be out of range in a statement that stores into a subscripted variable. The on-line user can test each subscript (following the RDY TO ABORT message) using the PRINT command. In many instances, the user may make temporary repairs via debug commands and continue execution after issuing a RESTART command or a GOTO command.

2. A CALL or function reference may have called a missing routine. The on-line user may be able to continue debugging (see 1, above) and bypass this problem by issuing SKIP commands or by attaching a GOTO command to an ON CALL command naming the missing routine.

3. A receiving sequence dummy may have been referenced without being properly set up. This can happen in the following ways:

   a. The calling sequence did not contain a sufficient number of arguments. (This produces the debugger message ARG # ERR; but if the program continues running, one of the trap messages may result). The on-line user may bypass this problem (similar to 2, above).

   b. The user may have branched into the middle of a subprogram without going through a normal entry point. The on-line user may back up and enter the subprogram via a GOTO command that specifies a CALL statement or a statement making a function reference to that subprogram.

   c. An ENTRY statement may have been called prior to a call on the subprogram where the referenced dummy did not appear in that ENTRY statement. Since the dummy did appear in the subprogram's FUNCTION or SUBROUTINE statement, the on-line user may get that dummy set up by taking action as in 3.b, above.

4. The program may have read into a Hollerith FORMAT statement, but the user neglected to specify the NMP (no memory protection) compilation option. The on-line user may bypass this problem by using SKIP or GOTO commands to prevent reading and referencing that FORMAT statement.

It should be clearly understood that the above suggestions to the on-line user do not correct error situations. Recompilation must be done to make corrections. The suggestions merely indicate ways in which the debugging run can be continued in an attempt to find further errors before recompiling. These suggestions also apply to the other FORTRAN run-time errors.

Monitor Error Messages

Any error message that is neither a debugger message nor a FORTRAN run-time error message must be a monitor error message. For a description of these messages, see the reference manual for the appropriate monitor.[†]

When the monitor displays one of its error messages, it also terminates the run. The debugger does not regain control. Fortunately monitor errors occur infrequently. In the event of a monitor error message, the on-line user does have one option. He may retry the run (for instance, by reloading) using debugger commands to locate the point at which the monitor error message occurs. Through judicious use of FLOW, AT, ON CALL (or ON CALLS), and step commands, the user can follow the progress of his run up to the point of the error. Once he has pinpointed the problem, the user may try again in order to bypass that problem (using SKIP or GOTO commands) so that remaining errors can be detected.

[†]XDS Sigma 5/7 Batch Time-Sharing Monitor (BTM) Reference Manual (Publication No. 90 15 77), XDS Sigma 7 Universal Time-Sharing Manual (Publication No. 90 09 07), XDS Sigma 5/7 Batch Processing Monitor (BPM) Reference Manual (Publication No. 90 09 54).

APPENDIX B. BATCH USAGE

In batch operation there is one GO command, and it must be the last debugging command. Data (if any) for the execution immediately follows the GO command record. Examples of the use of batch commands are shown in Figures 5 through 9. Commands may appear anywhere in columns 1-72.

```
GO
```

This setup provides the following automatic debugging features:

Argument-type checking.

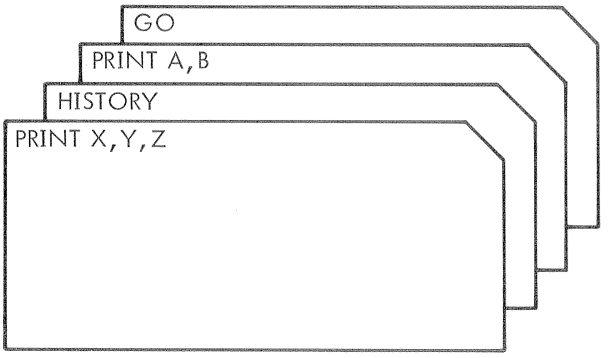
Argument-number checking.

Argument-protection checking.

Checking of stores into protected dummies (not available for FORTRAN IV-H).

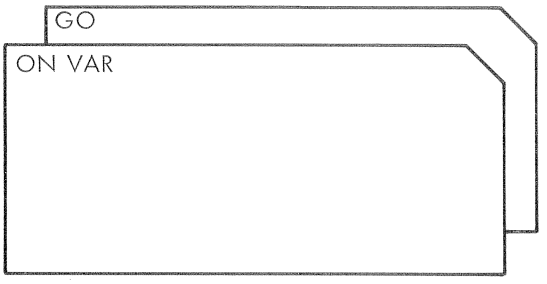
Run-time error messages containing the latest known position in the program.

Figure 5. Batch Usage – Automatics Checks Only



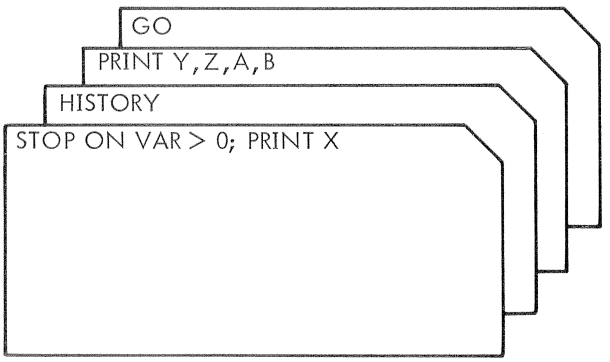
This setup provides two additional features whenever execution stops (with the debugger in control): display of the current history transactions and display of X, Y, Z, A, and B. This is independent of the use of DUMP, PDUMP, PMD, and PMDI. (DUMP and PDUMP are discussed under "Programs and Subprograms in the XDS Sigma 5/7 Extended FORTRAN IV Reference Manual, Publication No. 90 09 56; and PMD and PMDI are described under "Control Commands" in the XDS Sigma 5/7 Batch Processing Monitor (BPM) Reference Manual, Publication No. 90 09 54.)

Figure 6. Batch Usage – Postmortems



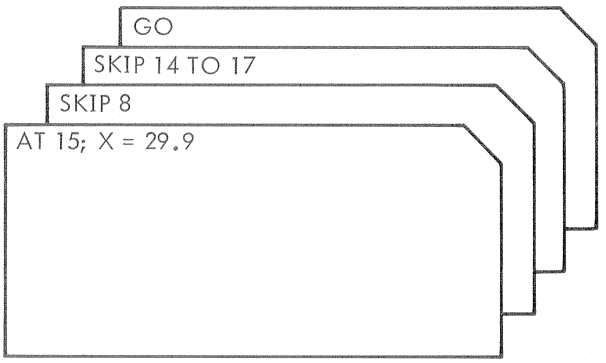
In addition to the automatic checks, this setup provides a display of VAR each time a data check-in occurs for the locations occupied by VAR.

Figure 7. Batch Usage – Trace of a Variable



This example is similar to the postmortem example (Figure 6), but in this case execution stops immediately if a data check-in detects VAR > 0. The following information is then displayed: the value of VAR, X, Y, Z, A, and B, and the history record contents.

Figure 8. Batch Usage – Trapping an Anomaly



This setup prevents execution of the statement code for source line numbers 8 and 14-17; but whenever line number 15 would have been executed, X is assigned the value 29.9. (The statement break is displayed.)

Figure 9. Batch Usage – Fixing a Simple Error

INDEX

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Xerox Data Systems

READER COMMENT FORM

XEROX®

We would appreciate your comments and suggestions for improving this publication.

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Operating
☐ Reference ☐ Maintaining ☐ Sales

Is the material presented effectively?

☐ Fully covered ☐ Well illustrated
☐ Clear ☐ Well organized

What is your overall rating of this publication?

☐ Very good ☐ Fair ☐ Very poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, please use the XDS Software Improvement or Difficulty Report (1188) instead of this form.

Thank you for your interest.

Fold and fasten as shown on back.
No postage needed if mailed in U.S.A.

Your name and return address.

STAPLE

STAPLE

FOLD

FIRST CLASS
PERMIT NO. 229
EL SEGUNDO, CALIF.

BUSINESS REPLY MAIL

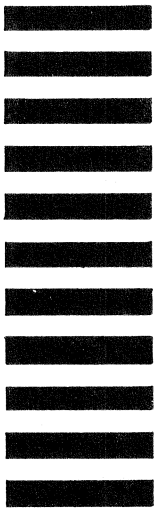
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

Xerox Data Systems

701 South Aviation Boulevard
El Segundo, California 90245

ATTN: PROGRAMMING PUBLICATIONS



CUT ALONG LINE

FOLD