

September, 1971

CORRECTIONS TO XEROX FORTRAN DEBUG PACKAGE (FDP)/REFERENCE MANUAL (Sigma 5-9 Computers)

PUBLICATION NO. 90 16 77A, JULY 1970

This revision package contains replacement pages for the inside front and back covers, and pages iii, iv, 1, 2, 5-10, 17-20, 23-32, 35-46, 57, 58, 63-65. These changes will be incorporated into the next revision of the manual.

Revision bars in the margins of replacement pages identify changes. Pages without the number 90 16 77A-1(9/71) at the bottom are included only as backup pages; any revision bars on such pages identify changes made in a previous revision.

CONTENTS

1.	INTRODUCTION	1			
	Batch and On-Line Debugging Capabilities	1			Position
	Input/Output	2			Source Line Number
					Statement Labels
					Offsets
					Qualifiers
2.	TYPICAL USE OF DEBUGGING COMMANDS	3	5.	DESCRIPTION OF COMMANDS	25
	GO Command	3		Stored Commands	25
	RESTART Command	3		SKIP Command	26
	REWIND Command	3		AT Command	26
	QUIT Command	4		ON Command	27
	ABORT LEVEL Command	4		ON CALL Command	28
	Step Command	4		ON CALLS Command	28
	Break Command	4		Attachable Commands	28
	SKIP Command	5		PRINT Command and OUTPUT Command	
	AT Command	6		and ? Command	29
	ON Command	8		Postmortem PRINT	30
	ON CALL Command	10		Value Display	30
	ON CALLS Command	10		Value Change Command	31
	PRINT Command	10		GOTO Command	31
	Value Change Command	11		FLOW and NOFLOW Commands	32
	GOTO Command	11		HISTORY and RESET HISTORY Commands	33
	FLOW Command	11		Postmortem HISTORY	33
	NOFLOW Command	12		USE FILE and USE ME Commands	34
	HISTORY Command	12		KILL Command	34
	RESET HISTORY Command	12		Direct Commands	35
	USE FILE Command	13		Single and Double Break Commands	35
	USE ME Command	13		LOC Command	36
	KILL Command	13		GO Command	36
				QUIT Command	36
				RESTART Command	36
				REWIND Command	37
				ABORT LEVEL Command	37
3.	DEBUGGER INTERFACING	15		Stepping and Backtracking Commands	37
	Debug Table	15		Stepping	38
	Name List	15		Backtracking	38
	Source Line Table	15		Error Detection Features	38
	Statement Label Table	16		Execution Stops	40
	Entry Point Names	16	6.	OPERATIONS	41
	Special Calls	16		Universal Time-Sharing Monitor (UTS)	41
	Initialization Call	16		Logging On	41
	Statement Check-In Calls	16		Compiling	42
	Data Check-In Calls	17		Loading	43
	Calling Sequence Calls	17		Executing	43
	Entry Point Calls	17		Interrupting, Stopping, and Logging Off	44
				Gaining Access to the Use-File	44
4.	DEBUGGER COMMAND LANGUAGE	18		Batch Time-Sharing Monitor (BTM)	44
	General	18		Logging On	44
	Typographical Conventions Used in This Manual	18		Compiling	46
	Common Command Elements	19		Loading	46
	Variable	19		Executing	46
	Array Element	19		Interrupting, Stopping, and Logging Off	46
	Whole Array	20		Gaining Access to the Use-File	47
	Scalar	20		Batch Processing Monitor (BPM)	47
	Constant	20		Compiling	47
	Argument	21		Loading	49
				Executing	49
				Use of FDP: UTS Versus BTM	49

7. RESTRICTIONS AND LIMITATIONS	50
Length of Command Input Line _____	50
Range of Source Line Numbers _____	50
Overlays _____	50
Not Available for Real-Time Runs _____	50
Nondebug-Mode Subprograms and Assembly Code _____	50
Output Constraints _____	50
Length of Execution _____	50
Program Size _____	50
INDEX	63

APPENDIXES

A. INFORMATION MESSAGES AND ERROR MESSAGES	51
Debugger Messages _____	51
Input/Output Error Messages _____	51
Status Messages _____	52
Position Messages _____	52
Execution Error Messages _____	52
Warning Messages _____	54
Command Error Messages _____	54
Immediate Errors _____	55
Activation Errors _____	55
Exercise Errors _____	56

FORTRAN Run-Time Error Messages _____	58
Monitor Error Messages _____	60

B. BATCH USAGE	61
----------------	----

FIGURES

1. Example of a Simple FORTRAN IV On-Line Program Run Under UTS _____	42
2. Example of a Simple FORTRAN IV-H Program Run Under BTM _____	45
3. FORTRAN IV Deck Setup for Debug-Mode Batch Processing _____	48
4. FORTRAN IV-H Deck Setup for Debug-Mode Batch Processing _____	48
5. Batch Usage – Automatic Checks Only _____	61
6. Batch Usage – Postmortems _____	61
7. Batch Usage – Trace of a Variable _____	61
8. Batch Usage – Trapping an Anomaly _____	62
9. Batch Usage – Fixing a Simple Error _____	62

TABLES

1. PRINT Commands _____	30
2. Debug Input/Output Error Messages _____	51
3. Debug Status Messages _____	53
4. Debug Execution Error Messages _____	53
5. Debug Warning Messages _____	55
6. Debug Command Error Messages _____	57

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
 - h. Locating the first word of a FORTRAN statement

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.

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 I/O 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). (In place of PRINT the user may also use OUTPUT or simply a question mark.) 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
@
```

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, S\$WTCH, 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 ($\textcircled{\text{RET}}$).

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 $\text{\textcircled{RET}}$ is used for carriage return, new line, or line feed. In on-line operation, commands are always terminated by $\text{\textcircled{RET}}$. 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). (The carriage return character is replaced by a new line character when debugging commands are interpreted.)
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 $\text{\textcircled{RET}}$ also delimit words and specifications. (The horizontal tab character is replaced by 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.

Although individual subscripts are not range-checked, the debugger normally verifies that array elements (in debug commands) are within the range of the array. The user can prevent this verification, however, by placing an "@" after the left parenthesis of the array element; for example

M(@5)

could be used to reference the item following the 4-element array M.

[†]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:

<u>No. of Characters</u>	<u>Type of Variable Affected</u>
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.

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

<u>Statement and Action Desired</u>	<u>Possible Commands</u>	
	<u>With State- ment Label</u>	<u>With Source Line Number</u>
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:

<u>Source Line No.</u>	<u>Label</u>	<u>Statement</u>
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 not 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 not 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 point 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 an entry point 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] . . . ^(RET)

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] ^(RET)

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 after 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.[†] 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^{††} 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.

[†]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.

^{††}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
? (question mark and OUTPUT are synonymous with PRINT)
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)

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 for 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).

[†]See "Stored Commands" above.

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 and ? Command

PRINT, OUTPUT, and the question mark are synonymous. 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, subarrays, arguments, and positions. A subarray is a contiguous portion of an array. In form, it resembles an array element, but at least one subscript is replaced by a "subscript-range" (two subscripts separated by a colon). As an example

```
PRINT A(2:3,4:5)
```

results in the display of

```
A(2,4)
A(3,4)
A(2,5)
A(3,5)
```

Subarrays are acceptable only in PRINT commands; they are not allowed in ON commands or value change commands.

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 ]
subarray [ 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.

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.	

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).

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, subarrays, 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 designed 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. (The debugger does not modify these characters even if they are nonprinting or unused EBCDIC characters.) 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

$$\left. \begin{array}{l} \text{variable} \\ \text{argument} \end{array} \right\} = \text{constant}$$

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 if 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.

^tSee Chapter 3.

^{tt}See "Position Messages" in Appendix A.

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
LOC
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 would be particularly evident during high volume displays, so a special procedure is followed in such cases. If a break is issued during the printout of one of the following, that printout is discontinued.

1. All variables in the program.
2. A whole array.
3. A subarray.
4. All arguments of some call.

The message "BROKEN" is displayed. If the PRINT command is attached to a stored command, execution resumes until normal recognition of the single break (thus, more than one "BROKEN" message may appear. However, if the PRINT command is not attached, the debugger takes the shortcut of immediately requesting another command.

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 I/O.

LOC Command

LOC is a direct command. Attachments are not allowed. The LOC command may only be used in on-line runs. It allows the user to find the hexadecimal location of the code occupied by a FORTRAN statement. The form of the LOC command is

```
LOC position [,position]. . . Ⓒ
```

This command is provided as a convenience when simultaneously using FDP and the machine-language debugger, DELTA. The hexadecimal value of a position (which is the location of its statement check-in call) is a useful reference point for machine-language debugging.

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]. . . (RET)
```

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 n th 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 Monitor.

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

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.

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

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

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" (see Appendix A) 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.

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 $\text{\textcircled{R}}$ 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 ($\text{\textcircled{C}}$) 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.

[†]For FORTRAN IV-H, resuming execution is pointless and an abort message occurs. See Appendix A.

^{††}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 4	>
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.

<pre> UTS AT YOUR SERVICE ON AT 03:43 MAR 19, '75 LOGON PLEASE: PAT,SANDY (RET) </pre>	} Logging on
<pre> !EDIT (RET) EDIT HERE *BUILD SOURCE (RET) 1.000 * READ(105,10) I (RET) 2.000 *10 FORMAT(G) (RET) 3.000 * STOP (RET) 4.000 * END (RET) 5.000 * (RET) *END (RET) </pre>	} Building a source file file via Edit
<pre> !COMMENT ON ME (RET) </pre>	Assigning DO (diagnostic output) to user's console
<pre> !SET M:SI DC/SOURCE (RET) </pre>	Assigning SI to a file
<pre> !FORT4 (RET) OPTIONS> DEBUG (RET) HIGHEST ERROR SEVERITY: 0 (NO ERRORS) </pre>	} Compiling in debug mode
<pre> !SET M:SI UC (RET) </pre>	Reassigning SI to the user's console
<pre> !LINK \$ ON LMNFILE (PO) (RET) LINKING \$ </pre>	} Linking the ROM output from the GO file
<pre> !START LMNFILE (RET) @STOP AT 3 (RET) @GO (RET) ?44 (RET) /3: @GO (RET) *STOP* 0 3: RDY TO STOP @GO (RET) STOPPING </pre>	Initiating execution User request for stop when source line 3 is reached User request for program to start Program request for input (caused by READ); user supplies 44 Stop produced by STOP AT 3 command above User request for program to continue Message produced by source line 3 (see SOURCE file above) Message produced by debugger User request for program to continue Message produced by debugger
<pre> !OFF </pre>	} Logging off
<pre> CPU = .0323 CON= :06 INT = 22 CHG = .0000 </pre>	

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 (REF)
```

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 PO 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 PO or FDP reference "loads" the public library routines (PI) plus the debug routines. A simple example of the LINK command is

```
!LINK (FDP) (REF)
```

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] (REF)
!RUN [mfl] [,mfl] . . . [,mfl] { (PO)
                               (FDP) } (REF)
```

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 PO 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 PO 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) (REF)
LINKING $
!START LMNFILE (REF)
```

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

```
!RUN $ ON LMNFILE (FDP) (RET)
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 (RET)
```

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 (RET)
```

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: acct, name [,pass] (RET)
!
```

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

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

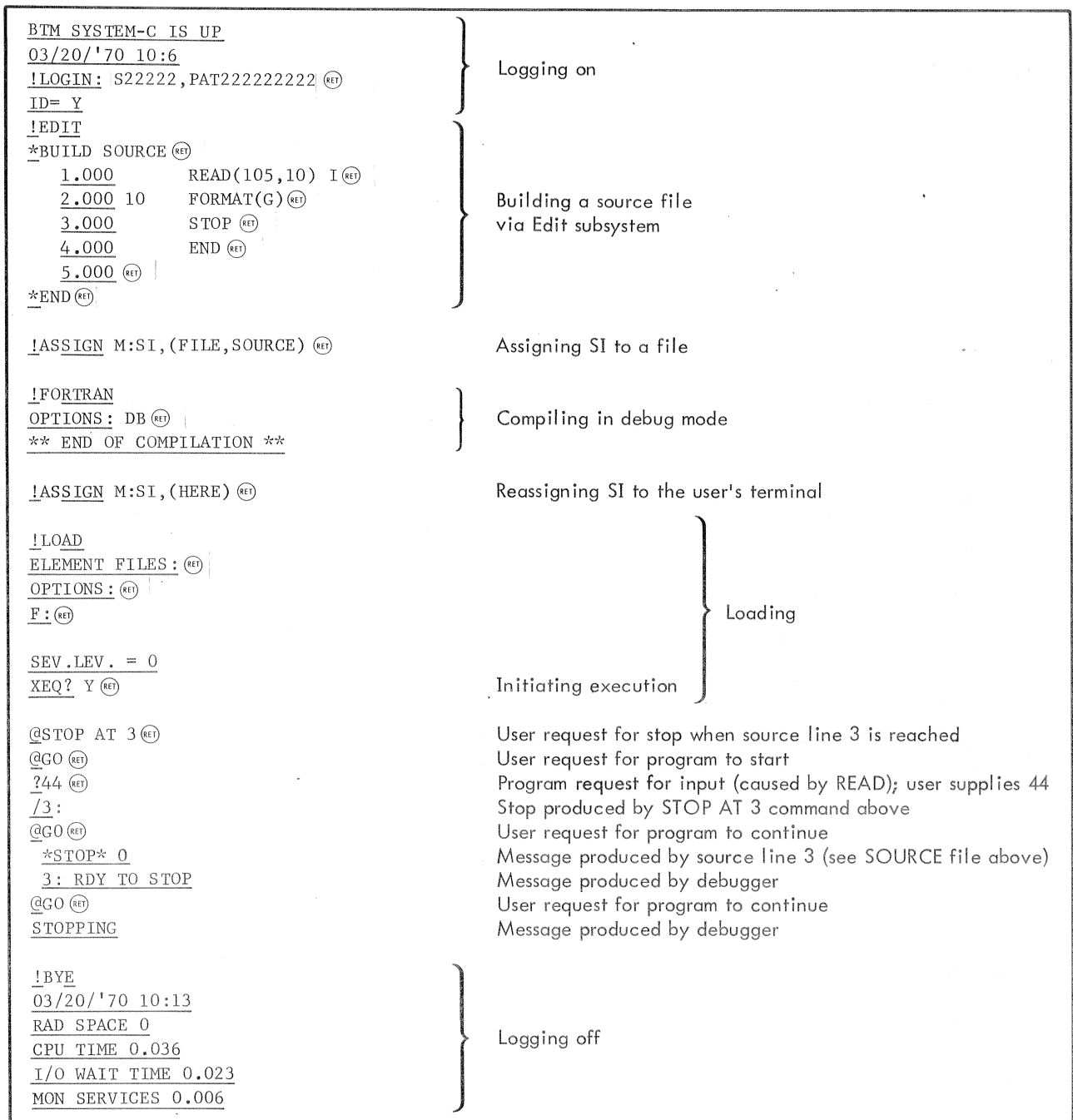


Figure 2. Example of a Simple FORTRAN IV-H Program Run Under BTM

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

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.
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 some 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* may be used in case of run-time 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.

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.

/, 23
? command, 29
@, 2
RET, 18

A

ABORT LEVEL command, 4, 37
activation errors, 55
active, 25
argument, 21
array element, 19
assigning SI (source input) to terminal, 43, 46
AT command, 6, 26
attachable command, 28, 24, 25
attachment errors, 56
attachments, commands that allow, 8
attachments, interpretation of, 24
automatic checks, 52

B

backtrack command, 37
backtracking through program flow, 12, 33
batch debugging capabilities, 1
batch operations (under UTS), 41, 44
Batch Processing Monitor (BPM) operations, 47
Batch Time-Sharing Monitor (BTM) operations, 44
batch usage examples, 61
blanks, 18
braces, 18
brackets, 18
break, definition of, 19
break command, 4, 35
BREAK key, 4, 35
breaks in UTS versus BTM, 49

C

calling sequence calls, 17
calls, 16
capabilities available in batch and on-line mode, 1
carriage return symbol, 18
check-in calls, 16
checks, 52
CMPX, 54
command elements, 19
command error messages, 54
command input line length, 50
command language, 18
commands
 ABORT LEVEL, 4, 37
 AT, 6, 26
 backtrack, 37
 break, 4, 35
 FLOW, 11, 32

GO, 3, 36
GOTO, 11, 31
HISTORY, 12, 33, 37
KILL, 13, 34
NOFLOW, 12, 32
ON, 8, 27
ON CALL, 10, 28
ON CALLS, 28
OUTPUT, 29
PRINT, 10, 29
QUIT, 4, 36
RESET HISTORY, 12, 33
RESTART, 3, 36
REWIND, 3, 36
SKIP, 5, 26
 step, 4, 37
USE FILE, 13, 34, 44, 47
USE ME, 13, 34
 value change, 11, 31
commands that allow attachments, 8
commands, description of, 25
commands, typical use of, 3
compilation, 1
compiling in debug mode, 15
 under BPM, 47
 under BTM, 46
 under UTS, 42
conditional ON command, 27
constant, 20
conventions, typographical, 18

D

data break, 19, 27
data check-in calls, 17
DB option, 43, 46, 47
debug commands, description of, 25, 24, 29
debug commands, typical use of, 3
DEBUG option, 42, 47
debug table, 15
debug table, search of, 24
debug-mode compilation, 15
 under BPM, 47
 under BTM, 46
 under UTS, 42
debugger interfacing, 15
debugger messages, 51
debugging capabilities, 1
deck setup for debug-mode batch processing, 48
deferred, 25
delimiters, 18
DELTA in UTS versus BTM, 49
detection of errors, 55
diagnostic output file (M:DO), 2
diagnostics, 51
direct command, 35, 24, 25

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.

directly used, 28
displaying values (PRINT), 10,30
DOUB, 54
double break, 4,35

E

element count, 19
elements, command, 19
entries in source line table, 15
entry identifier, 23
entry point calls, 17
entry point names, 16
error checks, 39
error messages, 51
error severity level, 37
error-detection features, 38,55
execution error messages, 52,39
execution of programs
 under BPM, 49
 under BTM, 46
 under UTS, 43
execution stops, 40
 under BTM, 46
 under UTS, 44
execution time limitation, 50
exercise errors, 56

F

FDP (FORTRAN debug package), 1
FDP option, 43
FDP usage in UTS versus BTM, 49
file retrieval, 44, 47
FLOW command, 11,32
flow trace, 11,32
FORTRAN debug package (FDP), 1
FORTRAN IV and IV-H, differences between, 1,6,15,22,
 39,41,44,47,61
FORTRAN run-time error messages, 58
function identifier, 23

G

global label, 6,22
GO command, 3,36
GO command, examples of batch usage, 61
GOTO command, 11,31

H

hexadecimal constant, 21
hexadecimal value, 31
HISTORY command, 12,33,37

I

identifier, 18,30
IF statement, 5
immediate errors, 55

information messages, 51
initialization call, 16
input/output, 2
input/output error messages, 51
interfacing, 15
INTG, 54
introduction to FDP, 1

K

KILL command, 13,34
KMPX, 54

L

label, 22
length of command input line, 50
length of execution, 50
limitations, 50
line feed, 18
linking debug tables, 17
loading programs
 under BPM, 49
 under BTM, 46
 under UTS, 43
LOC command, 36
local label, 22
logical constant, 20
LOGL, 54

M

M:DO, 2
M:SI, 2,43
messages, 51
missed kill, 35
monitor error messages, 60

N

name list, 15
natural number, 19
new line symbol, 18
NOFLOW command, 12,32
nondebug-mode assembly code, 50
nondebug-mode subprogram, 50
nonvector array, 19

O

offset, 22
ON CALL command, 10,28
ON CALLS command, 10,28
ON command, 8,27
on-line debugging capabilities, 1
on-line operations
 under BTM, 44
 under UTS, 41
operation procedures, 41
ordinary value, 31
output, 34

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.

OUTPUT command, 29
output constraint, 50
overlay restriction, 50

P

P0 option, 43
position, 6, 21
position messages, 52
postmortem HISTORY, 33
postmortem PRINT, 30
PRINT command, 10, 29
program size limitation, 50
prompt character (@), 2
prompt characters used in debugging, 41, 45
protection bit, 54
protection mismatch, 39

Q

qualified and unqualified items, interpretation of, 24
qualifier, 23, 6
qualifiers, roles for using, 24
question mark command, 29
QUIT command, 4, 36

R

range of source line numbers, 50
real-time run, 50
referencing an array, 19
relational operator, 9
RESET HISTORY command, 12, 33
RESTART command, 3, 36
restrictions, 50
revoking commands, 13, 34
REWIND command, 3, 37
run stops, 40
run-time error messages, 58

S

"S in column 1" statement, 15, 50
scalar, 20
semicolon in attachable command, 29
single break, 4, 35
size of program, 50
SKIP command, 5, 26
slash (/), 23
SNGL, 54
source input file (M:SI), 2, 43, 46
source line check-in calls, 16
source line number, 15, 21
source line number range limitation, 50

source line number with offset, 22
source line table, 15
special calls, 16
statement break, definition of, 19
statement check-in calls, 16
statement label, 22
statement label table, 16
statement label with offset, 22
status messages, 52
step command, 4, 37
STOP specification, 7, 26
stored command, 25, 24
string value, 30
subroutine identifier, 23
subscripting, 19
syntax of debug commands, 25

T

table, debug, 15
table, source line, 15
table, statement label, 16
text constant, 20
tracing program flow, 11, 32
type, 54
type incompatibility testing, 39, 54
type messages, 54
typographical conventions, 18

U

unconditional ON command, 27
unique identification, 25
Universal Time-Sharing Monitor (UTS) operations, 41
USE FILE command, 13, 34, 44, 47
USE ME command, 13, 34
use-file, gaining access to
 under BTM, 47
 under UTS, 44
UTS versus BTM, 49

V

value change command, 11, 31
value display (PRINT command), 10, 30
variable, 19
vector array, 19

W

warning messages, 54
whole array, 20

