File No. 7090-27
GC28-6393-2

IBM Systems Reference Library

# IBM 7090/7094 IBSYS Operating System
# Version 13
# IBJOB Processor Debugging Package

This publication describes the IBJOB Processor Debugging Package for use with the IBSYS Operating System (Version 13). The debugging package is a programming aid that enables the user to obtain dynamic dumps of specified areas of core storage and machine registers during program execution. The package contains two separate facilities: Compile-Time Debugging for COBOL programs and Load-Time Debugging for FORTRAN IV and MAP programs. Load-time debug requests are processed by the IBJOB Debugging Processor (IBDBL), #7090-PR-807. Compile-time debug requests are associated with the COBOL compiler (IBCBC).

## Preface

The IBM 7090/7094 IBJOB Processor Debugging Package provides a means of taking highly selective dumps of core storage areas and machine registers with a minimum of programming effort. By carefully selecting the areas to be dumped and the time at which to dump them, the user can obtain valuable information for locating and correcting program errors. The facilities described in this publication pertain to FORTRAN IV, COBOL, and MAP program debugging.

As a prerequisite to understanding this publication, the reader should be familiar with the IBJOB Processor, as described in the IBM publication *IBM 7090/7094 IBSYS Operating System: IBJOB Processor*, Form C28-6389, and with at least one of the programming languages accepted by the processor, as described in the IBM publications:

*IBM 7090/7094 IBSYS Operating System: Macro Assembly Program (MAP) Language*, Form C28-6392

*IBM 7090/7094 IBSYS Operating System: FORTRAN IV Language*, Form C28-6390

*IBM 7090/7094 IBSYS Operating System: COBOL Language*, Form C28-6391

The compile-time debugging language (for use with COBOL programs) is based on COBOL, and the load-time debugging language (for use with both FORTRAN IV and MAP programs) is based on FORTRAN IV. The MAP programmer who is totally unfamiliar with FORTRAN should be able to use all of the facilities described herein with limited reference to the FORTRAN language publication listed above.

The 7090/7094 IBJOB Processor Debugging Package requires the same minimum machine configuration as the 7090/7094 IBJOB Processor except that a unit specified as SYSCK2 is required for debugging output when the load-time debugging facility is used.

The problem of locating errors in programs rapidly and efficiently is of major concern to all computer users. The 7090/7094 IBJOB Processor Debugging Package meets this problem by allowing the programmer to manipulate data, control processing, or dump the contents of any relevant areas by inserting debug requests at key points in his program. To use the debugging package, the programmer writes a *debug request* in the appropriate debugging language. Each request specifies the point(s) in the program at which the specified action is to be taken. Any reasonable number of requests may be given for a single program.

The debugging package provides two types of debugging. The first, *compile-time debugging*, is included with IBCBC at compilation to specify dumps at various points in a COBOL source program. In this type, the text of debug requests is similar to the COBOL language. The second type, *load-time debugging*, uses the capabilities of IBLDR to provide debugging during the execution of a FORTRAN IV or MAP program without recompilation or reassembly. In this type, the text of debug requests is written in form similar to the FORTRAN IV language.

This publication describes the debugging package in two parts: Part I describes the compile-time facility for COBOL programs; Part II describes the load-time facility for FORTRAN IV and MAP programs. These parts are independent of each other, so that reference to one is not required when reading the other. The material in Part II is divided into two sections, "Load-Time Debug Requests" and "Additional Load-Time Debugging Features." The FORTRAN IV programmer will generally use only the facilities described in the first section, whereas the MAP programmer will use the facilities described in both sections.

The following conventions apply to all card formats given in this publication:

1. Brackets [ ] indicate that the enclosed material may be omitted.
2. Braces { } indicate that a choice of the enclosed material is to be made by the user.
3. Upper-case words, if used, must be present in the form indicated.
4. Lower-case words represent generic quantities whose values must be supplied by the user.

MAJOR REVISION

This edition, Form C28-6393-2, is a major revision of the previous edition, Form C28-6393-1. It clarifies the format of the NAME statement and the ON statement, corrects minor errors, and introduces new material.

Forms C28-6393-0, C28-6393-1, N28-0164-0, and N28-0170-0 are now obsolete.

Additions or changes are indicated by a vertical line to the left of the text; new or revised figures are denoted by the symbol • to the left of the caption.

# Contents

# Part I: Compile-Time Debugging for COBOL Programs

The compile-time facility of the debugging package enables the COBOL programmer to specify debug requests with his source-language program. The requests are compiled with the source program and are executed at object time. The text of the request is very similar to the procedural text of COBOL. In addition, a special count-conditional statement is provided. Since procedural capabilities of the COBOL compiler are available, a user can be highly selective in specifying what is to be dumped. He can manipulate and test the values of data items in his program and dump only pertinent and meaningful information, without affecting execution of the program itself.

## Compile-Time Debugging Packet

All compile-time requests for a given program are grouped together into a *debugging packet.* The compile-time debugging packet is placed *immediately following* the $CBEND card of the associated source program.

### Compile-Time Debug Requests

Each compile-time debug request is headed by the control card $IBDBC. The $IBDBC card serves two functions: it identifies individual requests, and it defines the point at which the request is to be executed. The general form of this card is

```
1        8        16-72
$IBDBC   [name]   location   [, FATAL]
```

where the parameters are described as follows:

| | |
|---|---|
| name | An optional user-assigned control section name, which permits deletion of the request at load time. This name must be a unique control section name consisting of up to six alphabetic and/or numeric characters, at least one of which must be alphabetic. |
| location | The COBOL section-name or paragraph-name (qualified, if necessary) indicating the point in the program at which the request is to be executed. Effectively, debug request statements are performed as if they were physically placed in the source program following the section- or paragraph-name, but preceding the text associated with the name. |
| FATAL | If this option is exercised, loading and execution of the object program will be prevented whenever an error of level E or greater occurs *within* a debug request statement. If FATAL is not specified, a COBOL error of level E or less, when encountered in the procedural text of a debug request, will not prevent loading and execution of the object program. Instead, an attempt will be made to interpret the statement. If interpretation is impossible, the erroneous statement (but not the entire request if it consists of more than one statement) will be discarded. |

The text of the debug request follows immediately after the $IBDBC card. The text may consist of any valid procedural statements conforming to the requirements of the COBOL language and format and of the count-conditional statement described in the following text. The only restriction on these statements is that they may not transfer control outside of the debug request itself. A procedure-name in a debug request must be unique to the request in which it appears, to all other debug requests, and to the source program. Display statements in a debug request write on SYSOU1.

A compile-time debug request is terminated by an end-of-file card, another $IBDBC card, or a $-control card.

#### Count-Conditional Statement

A count-conditional statement, available for use only in debug requests, provides the programmer with a means of qualifying the time when a debugging action should be taken. The count-conditional statement has the same structure as the COBOL IF statement (conditional, true option, false option) and may be used in the same manner; i.e., it may be nested within other count-conditional or IF statements and may have other count-conditional or IF statements nested within it. The general form of the count-conditional statement is

$$\text{ON } n_1 \quad [\text{AND EVERY } n_2] \quad [\text{UNTIL } n_3] \quad \text{statement-1}$$

$$\left[\begin{Bmatrix}\text{ELSE}\\ \text{OTHERWISE}\end{Bmatrix} \quad \text{statement-2}\right]$$

where $n_1$, $n_2$, and $n_3$ are positive integers. If the AND EVERY $n_2$ option is not specified, but the UNTIL $n_3$ option is specified, $n_2$ is assumed to be one. The UNTIL option means *up to but not including* the $n_3$th time.

Some examples of the count-conditional statement follow:

ON 3 DISPLAY A.
: On the third time through the count-conditional statement, A is displayed. No action is taken at any other time.

ON 4 UNTIL 8 DISPLAY A.
: On the fourth time through the seventh time, A is displayed. No action is taken at any other time. (This example implies, and has the same effect as, the statement ON 4 AND EVERY 1 UNTIL 8 DISPLAY A.)

ON 5 AND EVERY 3 UNTIL 12 DISPLAY A.
: On the fifth, eighth, and eleventh times through the

count-conditional statement, A is displayed. No action is taken at any other time.

ON 3 AND EVERY 2 DISPLAY A.

A is displayed on the third, fifth, seventh, ninth, etc., times. On the first, second, fourth, sixth, etc., times, no action is taken.

ON 2 AND EVERY 2 UNTIL 10 DISPLAY A ELSE DISPLAY B

On the second, fourth, sixth, and eighth times, A is displayed. B is displayed at all other times.

### *A Compile-Time Debugging Packet*

The numbers given in the left-hand column of the example in Figure 1 are for purposes of reference in the explanations that follow; they are not part of the requests themselves. The numbers in the first line across indicate the card columns in which the various fields begin.

```
1      8   12  16

$IBDBC         A
           ON 1 AND EVERY 3 UNTIL 8 DISPLAY
           'Z=' Z.
$IBDBC CNTRL   B OF C
           IF S UNEQUAL TO T DISPLAY 'S=' S,
           MOVE T TO S. DISPLAY 'T=' T.
$IBDBC         D, FATAL
           IF V GREATER THAN VMAX ON 1 UNTIL
           10 DISPLAY 'V OUT OF RANGE, V=' V
           ELSE STOP RUN.
```

Figure 1. Example of a Compile-Time Debugging Packet

In the first request, on the first, fourth, and seventh time that control passes through point A in the program, Z is displayed (in its own format) with the identifying heading Z =.

In the second request, the value of T (with the identifying heading T = ) is displayed at the point in the program identified as B OF C. Also, if S is unequal to T, S is also displayed and its value is adjusted to the value of T. If desired, this debug request may be deleted during this and/or subsequent runs by using a $OMIT control card, which is described in the publication *IBM 7090/7094 IBSYS Operating System: IBJOB Processor*, Form C28-6389.

Execution of the third request causes both the message V OUT OF RANGE, V = and the value of V to be displayed the first nine times that V is greater than VMAX when program control passes through point D. On the tenth time, this request causes an exit from the program (i.e., ELSE STOP RUN). The FATAL option on the $IBDBC card heading this request prohibits loading of the source program if the compiler encounters an error of level E or greater in the text of this request.

# Part II: Load-Time Debugging for FORTRAN IV and MAP Programs

The load-time facility of the debugging package provides FORTRAN IV and MAP programmers with the means to insert debug requests at load time to be executed with the object program. MAP object programs include those generated by IBCBC as well as those written in MAP itself. Thus, the COBOL programmer may, if desired, take advantage of the load-time facility and debug from an assembly listing of his program; the FORTRAN IV programmer may also use the load-time facility at the MAP level.

The debugging language used with the load-time facility is derived from the FORTRAN IV language, with changes and additions made for debugging purposes. The statements available in the debugging language permit the programmer a high degree of flexibility in obtaining meaningful data in his dumps. It is possible to perform arithmetic operations on object time and debug packet values, to test and manipulate results, and to select the quantities to be dumped. Also, the programmer can reference symbols appearing in the source program by selecting the appropriate dictionary option in his source program. When more than one name has been given to a program area (for example, by the use of equivalence statements or COMMON definitions in different decks), the load-time debugging postprocessor chooses unpredictably from among these names when dumping the area. The only way this problem can be avoided is by not multiply-labelling such areas; it can be minimized by using the DUMP 'msg' statement to label the dump as desired.

In the discussions that follow, the term "integer" is to be interpreted as follows: if it is prefixed with a leading zero and does not contain any invalid octal characters (i.e., 8 or 9), it will be considered octal. Otherwise, it will be considered decimal.

## Load-Time Debugging Packet

All of the load-time debug requests for a particular job run (which may consist of any configuration of FORTRAN, COBOL, or MAP source and/or object decks) are grouped together into what is called the *debugging packet*. The packet is headed by a $IBDBL card and terminated by an *DEND card. These control cards are described in the publication *IBM 7090/7094 IBSYS Operating System: IBJOB Processor*, Form C28-6389. The load-time debugging packet is placed at the beginning of the job deck, preceding the source and/or object decks.

### Load-Time Debug Requests

A debug request is a set of actions to be performed at an indicated point in the program called the insertion point. Each load-time debug request is headed by the *DEBUG card, which identifies individual requests and specifies the insertion point(s) of the request in the program.

The general form of the *DEBUG card is:

```
1        8            16-72
*DEBUG   [deckname]   loc1  [, loc2, loc3, . . .]
```

Blanks may be included in the variable field for legibility but they may not be imbedded. The parameters of the *DEBUG card are as follows:

| | |
|---|---|
| deckname | The name of the object deck to which this debug request applies. If this field is blank, the last deckname specified on the preceding *DEBUG or *REDEF card is assumed; if a deckname was not previously specified, the request is deleted. Library subroutines cannot be referred to in a debug request. |
| loc1, . . . | The location(s) of the executable instruction(s) at which this debug request is to be inserted. A location may be specified in any of the following ways: |

1. A statement number (FORTRAN only).
2. A symbol.[1] Symbols used in debug requests may not contain parentheses.[2]
3. A symbol ± an unsigned decimal integer.
4. =R followed by an unsigned octal integer for a relative location (i.e., relative to the load address of the deck).
5. =A followed by an unsigned octal integer for an absolute location.
6. An internal formula number of a FORTRAN statement with the suffix A (e.g., 10A).[3]

Because of the method by which insertions are made in the program (i.e., the STR instruction), the programmer should take care not to specify insertion points at instructions whose prefix might be modified during execution. This is primarily of concern to the MAP programmer. Violation of this rule may result in unpredictable results and/or actions. No check is made for this condition.

The debug request is executed as if it had been physically inserted in the deck at the specified location(s). The debug request action occurs before the execution of any action indicated by the statement or instruction at that location.

[1]These symbols must consist of one to six nonblank BCD characters. At least one nonnumeric character must be included, but none of the following ten characters may be used:

| | | | |
|---|---|---|---|
| + | (plus sign) | = | (equal sign) |
| — | (minus sign) | , | (comma) |
| * | (asterisk) | ' | (apostrophe) |
| / | (slash) | ( | (left parenthesis) |
| $ | (dollar sign) | ) | (right parenthesis) |

[2]This restriction applies only to symbols that are explicitly given in a request. Symbols of this type may appear in the debugging dictionary, and the *REDEF card (discussed later) provides a means of renaming them so that they may be used in requests.

[3]The internal formula number can be referenced only when the full debugging dictionary has been requested.

The variable field of an *DEBUG card may be extended over more than one card by punching cards following the first as shown:

```
1        8        16-72
*ETC              extension of variable field
```

Immediately following the *DEBUG card is the text of the request itself. If an invalid or erroneous action is specified in the text, that action is deleted. The text consists of procedural statements written in the FORTRAN format. At least one blank should follow each statement verb (e.g., DUMPbx).[3] These statements are derived from the FORTRAN IV language, with additions and changes made for debugging purposes. The permissible statements are:

STOP statements
PAUSE statements
CALL statements
RETURN statements
Unconditional GO TO statements
SET (arithmetic) statements
Logical IF statements
ON statements
DUMP statements
LIST statements
NAME statements

Comment cards having a C in column 1 are allowed. Library subroutines cannot be referred to in a debug statement.

The operator **(exponentiation) and functions are not allowed, nor are references to the dummy variables of functions and subroutines. Subscripted variables may have no more than three subscripts. FORTRAN variables having more than three dimensions are treated as single-dimension variables by the debugging package, that dimension being the product of the dimensions specified in FORTRAN. The following rules apply to the referencing of such variables:

1. The variable is treated as though it had occurred in the statement:

DIMENSION var(d)

instead of:

DIMENSION var( $d_1, d_2, \ldots, d_n$ )

where:

$n > 3$ and $d = d_1{*}d_2{*}\ldots{*}d_n$.

2. In order to refer to the element:

var $(s_1, s_2, \ldots, s_n)$ $(n > 3)$

the reference must be written as

var (s)

where:

$$s = s_1 + d_1 (s_2 - 1 + d_2 (s_3 - 1 + \ldots + d_{n-1} (s_n - 1) \ldots))$$

Thus, if we desire to refer to A(2, 3, 4, 5) and A has dimension (10, 9, 8, 7) we compute:

$$s = 2 + 10 (3 - 1 + 9 (4 - 1 + 8 (5 - 1)))$$

or

$$s = 2 + 10\cdot 2 + 10\cdot 9\cdot 3 + 10\cdot 9\cdot 8\cdot 4 = 3172$$

so that the desired reference is A(3172). The reference A(I, J, K, L) would be written:

A(I + 10*(J − 1 + 9*(K − 1 + 8* (L − 1))))

in the debugging language.

[3]However, unlike FORTRAN IV, the debugging routine treats blanks in a statement as terminators. Therefore, no blanks may be imbedded in a character string that is to be treated as a single symbol.

### Debugging Control Statements

The STOP statement terminates execution.

The PAUSE statement prints the message "DEBUG REQUEST PAUSE" and then the machine halts; pushing START restarts processing at the next debug statement.

The CALL statement is used in calling subroutines. It has the form:

CALL SUBR
or
CALL SUBR ( $arg_1, arg_2, \ldots$ )

The CALL cannot induce overlay and cannot have non-standard returns. Machine registers are saved upon entry to the CALL and restored upon exit from the CALL. Machine registers that are initially available in the CALL differ from those that are saved.

NOTE: A CALL should not be made to a subroutine which in turn executes another debug request. Such a relationship between debug requests is called "nesting." An error message will result from the attempted execution of the second debug request and execution of the entire job will terminate.

The RETURN statement causes a return to the interrupted program; there is an implicit RETURN at the end of each debug request.

The statement GO TO n, where n is a decimal integer, is an unconditional transfer to the debugging statement having the corresponding integer (statement number) n in columns 1-5. Statement numbers used in an unconditional GO TO statement must refer to statements within the debugging packet, *not* to statement numbers in the deck being debugged.

### SET (Arithmetic) Statement

The SET statement provides the programmer with arithmetic capabilities within a debug request. The general form of this statement is

SET s

where s is any valid FORTRAN IV arithmetic statement not containing functions, the **(exponentiation) operator, nor the logical constants .TRUE. and .FALSE.. Because FORTRAN IV conventions override MAP notation, those MAP symbols that would be incorrectly treated in an arithmetic statement (e.g., 1.2) should be redefined prior to use. See the explanation of the *REDEF card for details of the redefining facility.

### Logical IF Statement

The debugging logical IF statement is similar to that of FORTRAN IV, with certain additions and restrictions. The general form of this statement is:

IF (t) s
or
IFbtbs

where b represents one or more blanks, t is any logical expression not containing function calls or the ** operator, and s is an unconditional executable statement or an ON statement followed by an unconditional executable statement. The statement is executed if and only if it is logically true (nonzero).

The permissible logical operators (where b represents a blank and x and y are logical expressions) are:

| | | |
|---|---|---|
| bNOTbx | or | b .NOT. bx |
| xbANDby | or | xb .AND. by |
| xbORby | or | xb .OR. by |

The permissible relational operators are:

| RELATION | | | DEFINITION |
|---|---|---|---|
| bGTb | or | b .GT. b | Greater than |
| bGEb | or | b .GE. b | Greater than or equal to |
| bLTb | or | b .LT. b | Less than |
| bLEb | or | b .LE. b | Less than or equal to |
| bEQb | or | b .EQ. b | Equal to |
| bNEb | or | b .NE. b | Not equal to |
| bLGTb | or | b .LGT. b | Logically greater than |
| bLGEb | or | b .LGE. b | Logically greater than or equal to |
| bLLTb | or | b .LLT. b | Logically less than |
| bLLEb | or | b .LLE. b | Logically less than or equal to |
| bLEQb | or | b .LEQ. b | Logically equal to |
| bLNEb | or | b .LNE. b | Logically not equal to |

The permissible arithmetic operators are:

| | |
|---|---|
| + | addition |
| − | subtraction |
| * | multiplication |
| / | division |

Where parentheses are omitted, the hierarchy of operations is as follows: * and /; + and −; relational operations; NOT; AND; OR.

Following are some examples of the IF statement:

```
IF (B EQ A*3.5) DUMP X, Y, Z
IF A(I, I-J) EQ 3.4E2*QSUM DUMP X
IF (X .EQ. 3 .AND. Z .LT. 24) GO TO 3
IF LOGVAR .AND.
   (ALPHA+6 LE BETA OR LGVAR1) RETURN
```

### *ON Statement*

The ON statement is a count-conditional statement that permits the programmer to control the time when a debugging action is to be taken. It is similar to the FORTRAN IV logical IF statement and is of the general form

ON [(x)] $a_1$ [, $a_2$] [, $a_3$] s

where the $a_i$ are any arithmetic expressions (if they are not integral, they will be truncated); x is a unique symbol, which should not be contained in the debugging dictionary, that represents a counter name; and s is an unconditional executable statement or an IF statement followed by an unconditional executable statement. Additional information about the debugging dictionary is provided in the publication *IBM 7090/7094 IBSYS Operating System: IBJOB Processor*, Form C28-6389.

The ON statement is defined as true the $a_1$th time it is executed and every $a_3$th time thereafter until $a_2$ is exceeded. If $a_2$ is null, the statement is true the $a_1$th time and every $a_3$th time thereafter. If $a_3$ is omitted, it is assumed to be one. If both $a_2$ and $a_3$ are omitted, the statement is true only the $a_1$th time.

If x is specified, it creates a named counter for the ON statement and x may be used in any computation or test, the same as any other variable, and may be named in other ON statements. If the same counter is used by several ON statements, it is incremented for each one of the ON statements that is executed. Thus, the counter can be set and reset to any desired value. Effectively, the statement ON(CTR) $a_1$, $a_2$, $a_3$ s is the same as the statements

```
NAME   CTR/ = NEW(X)
       .
       .
       .
SET CTR = CTR + 1
IF (CTR GE a1 AND CTR LE a2
  AND ((CTR−a1)/a3) *a3 EQ CTR−a1)        s
```

All references to x are taken as references to this counter. Therefore, if x is duplicated by a symbol in the debugging dictionary, it will not be possible to refer to that object program symbol in a request.

If x is not specified, an unnamed counter is created internally. This counter is distinct from any other counter.

### *DUMP Statement*

The DUMP statement causes the dump of the quantities which are to be printed as debugging output. It is similar in structure to the FORTRAN IV WRITE statement and is of general form

DUMP List

where list is a series of items that are either direct references to the data to be dumped or the statement number(s) of LIST statements specifying the data to be dumped. The acceptable data specifications for either direct references or LIST statements are itemized under the discussion of the LIST statement.

The DUMP statement causes information to be written on SYSCK2. The postprocessor edits the data on SYSCK2 and writes it on SYSOU1. The formats supplied for the items of the DUMP statements are as follows:

1. The symbolic reference of the item along with its relative and absolute locations and deck name is written to identify the debugging output.

2. The value(s) of the item is written. The format, which is based upon the number of elements in the item and the mode of the item, is derived as follows:

| TYPE | NO./LINE | FORMAT |
|---|---|---|
| Octal | 4 | op$^4$ xxxxxx xxxxxx |
| Symbolic Instruction | 2 | ±x xxxxx x xxxxx op a,t,d$^4$ |

| TYPE | NO./LINE | FORMAT | |
|---|---|---|---|
| Symbolic Command | 2 | ±x xxxxx x xxxxx op(n*) a,,d[4] | |
| Floating-Point | 6 | ±.xxxxxxxx±xx | |
| Fixed-Point | 6 | ±xxxxxxxxxxxx. (leading zeros dropped) | |
| Double-Precision | 4 | ±.xxxxxxxxxxxxxxxxD±xx | |
| Complex | 3 | ±.xxxxxxxx±xx | ±.xxxxxxxx±xxJ |
| Logical | 8 | .TRUE. or .FALSE. | |
| Alphameric | 72 char | xxxx...xxx | |

### LIST Statement

The LIST statement specifies the storage areas and/or registers that are to be dumped; it is of the general form where the statement number is a standard FORTRAN statement number of up to five numeric characters (punched in columns 1-5) and the items denote the addresses of the quantities to be dumped. Any reasonable number of items may be specified; they are separated by commas. The permissible items are detailed in the following text. Except where otherwise indicated, the term "symbol" in the following items refers to a symbol that either appears in the debugging dictionary or is defined in a NAME statement. The mode of the data dumped is determined from the debugging dictionary or from the NAME statement; if this information is not available, the dumps will be octal except where the mode is specified as in item 2 in the following text.

The permissible items are:

1. quantity—This may be one of the following:

| | |
|---|---|
| symbol | This causes the array, the double-precision floating-point number, the complex number, or the single word denoted by this symbol to be dumped. |
| symbol (subscript(s)) | This causes the array element, the double-precision floating-point number, the complex number, or the single word denoted by this subscripted symbol to be dumped. Any symbol may be singly subscripted, but only those symbols that have been dimensioned may have more than one subscript. The subscripts may be any arithmetic expressions. The mode of the dump is the same as the mode of the symbol. If ALPHA(6) is specified, the contents of the location ALPHA+5 is dumped except where ALPHA is double precision or complex, in which case ALPHA+10 and ALPHA+11 are dumped. |
| symbol ± n | This causes the single word denoted by this quantity to be dumped. Symbol is as defined in the preceding text and n is an unsigned decimal integer. The location in core storage which is dumped is n places removed from the symbol. The mode of the dump is determined from the referenced location and is not necessarily the same mode as the symbol. |
| =Rn | This causes the word at relative location n, where n is an unsigned octal integer, to be dumped. Only one word is dumped, even though it may be part of a double-precision or a complex number. |
| =An | This causes the word at absolute location n, where n is an unsigned octal integer, to be dumped. Only one word is dumped, even though it may be part of a double-precision or a complex number. |

[4] op (n*) a, t, d refers to the symbolic representation of a machine language instruction. (n*) is x if this is a dump request point. This is primarily of interest to the MAP programmer.

The MAP programmer is referred to the sections "Quantities Available for Use in Debug Request Statements" and "Address Computation" for more complex ways of addressing quantities.

2. (locl, loc2 [, m]) – This causes the region from locl through loc2 to be dumped in the format m. Loc is any of the quantities previously defined or a statement number within the source program. If m is specified, it overrides any other mode designations given for the quantities involved. If m is not specified, the region is dumped in the mode(s) of the item(s) in the region. These modes are supplied in the debugging dictionary or NAME statement. The permissible values of m are:

| | |
|---|---|
| O | Octal |
| S | Symbolic instruction |
| C | Symbolic command |
| F | Floating-point number |
| X | Fixed-point number |
| D | Double-precision floating-point number |
| J | Complex number |
| L | Logical |
| H | Alphameric |

Thus, (LOC3(6), LOC4(3,4), F) causes LOC3 + 10 through LOC4+11 to be dumped in floating-point mode if LOC3 is a double-precision array and LOC4 is a floating-point array with dimensions of (3, 10). (6, =R127) causes the region from statement number 6 through relative location $127_8$ to be dumped in the mode(s) supplied by the debugging dictionary or NAME statement. (ARRAY1,ARRAY2,X) causes all of ARRAY1, all intervening locations, if any, and all of ARRAY2 to be dumped in the fixed-point decimal mode.

3. (data list, range) – This causes selected elements of arrays to be dumped. Data list may be one of the following:

| | |
|---|---|
| symbol (subscript(s)) | e.g., (A(I), I = 1, 4) |
| (data list, range) | e.g., ((B(I, J), I = 1, 4), J = 1, 3) |
| data list, data list, . . . | e.g., (A(I), (C(J, I), J = 1, 9, 2), D(I), I = 1, 4) |

Any arithmetic expression may be used as a subscript. If the expression is not integral, it will be truncated.

Range is of the form

$v = a_1, a_2 [, a_3]$

where v is any symbol and the $a_i$ are arithmetic expressions. If the same symbol appears elsewhere in the program or in another debug request, that use(s) will be ignored for this statement. The dumps specified by the data list are taken for the value of v, namely $a_1$ through $a_2$ in increments of $a_3$, or increments of one if $a_3$ is omitted. The maximum depth to which ranges may be nested is three. The following example of element specification is *invalid* because it contains four nested ranges:

((((A(I, J, K), B(I, J, L), I=1, 2), J=1, 2), K=1, 2), L=1, 2)

The following are valid examples of this element specification:

(A(I, I), I=1, 3)
  dumps $A_{11}$, $A_{22}$, $A_{33}$.
(B(I, 6−I), I=1, 5, 2)
  dumps $B_{15}$, $B_{33}$, $B_{51}$.
((A(I, J), I= 1, 3), J=6, 8, 2)
  dumps $A_{16}$, $A_{26}$, $A_{36}$, $A_{18}$, $A_{28}$, $A_{38}$.
((C(2*J−4, 10−3*I), I=1, 3), J=3, 4)
  dumps $C_{27}$, $C_{24}$, $C_{21}$, $C_{47}$, $C_{44}$, $C_{41}$.
(A(J, J), (C(2*J−4, 10−3*I), I=1, 3), J=3, 4)
  dumps $A_{33}$, $C_{27}$, $C_{24}$, $C_{21}$, $A_{44}$, $C_{47}$, $C_{44}$, $C_{41}$.

The statement

    DUMP(A(I, J), I=1, 4)

dumps A(1,J), A(2,J), A(3,J), and A(4,J), where J is a variable previously defined either in the source program or in a NAME statement.

4. // or /control section name/—This causes blank COMMON or the control section named to be dumped.

// ±n or /control section name/ ±n—This causes the location at the beginning of blank COMMON ±n or the location at the control section ±n to be dumped. FORTRAN labeled COMMON names are control section names.

5. PROGRAM—This causes the entire object program exclusive of library subroutines to be dumped.

6. 'msg'—This causes the message to be printed as it appears. The symbol msg refers to any message not containing a quotation mark; however, the external quotation marks are required.

7. CONSOLE – This causes the following items to be written: the contents of the accumulator, the multiplier-quotient register, the sense indicators, the index registers, as well as the status of the sense lights, the sense switches, the entry keys, and the trapping, overflow, divide check, and I-O check indicators.

To allow for interdeck communication, any item or set of items may be qualified with an associated deck name by preceding it with the deck name and two connecting dollar signs, as follows:

    deckname$$item
          or
    deckname$$(item, item, . . . .)

For example, A$$B refers to symbol B in deck A and the statement DUMP AB$$(BB,CB,(RB,SB,O)) dumps items BB, CB, and the octal region RB through SB in deck AB.

### NAME Statement

The NAME statement permits the programmer to define symbols for use within debug requests and to supply modal and/or dimensional information for the symbols. This definition overrides any other definition or any other associated information for the symbol within debugging text. The form of the NAME statement used primarily by the FORTRAN programmer is:

NAME symbol$_1$/=NEW [(mode [(dimensions)])]
[,symbol$_2$/=NEW [(mode[(dimensions)])] . . . ]

where the parameters are defined as follows:

| | |
|---|---|
| symbol | Any valid FORTRAN symbol. |
| =NEW | =NEW specifies that a location of octal format or an area corresponding to the specified mode and dimensions is generated for the symbol. |
| (mode) | Mode can be either X, F, D, J, H, or L. These are defined in the section "The List Statement." |
| (dimensions) | The dimensions are the same as the dimensions specified by a DIMENSION statement. (For an explanation of this statement and examples of it, see the section entitled "DIMENSION Statement" in the publication IBM 7090/7094 IBSYS Operating System: FORTRAN IV Language, Form C28-6390.) |

This form of the NAME statement may be used to establish variables for use in the debugging requests beyond those defined in the debugging dictionaries. The NAME statement in the request must precede any references to the symbol it defines. No more than three dimensions may be specified.

### Debugging Dictionary

The debugging dictionary is a communication device between the program that is being debugged and the debug package. This dictionary contains symbols and pertinent data about the symbols, such as their relative locations, their modes, and their dimensions. It also contains FORTRAN statement numbers and their relative locations. If requested by the programmer on a $IBMAP or a $IBFTC control card, the debugging dictionary is produced. Further information about the debugging dictionary is contained in the publication *IBM 7090/7094 IBSYS Operating System, IBJOB Processor,* Form C28-6389.

### A FORTRAN IV Load-Time Debugging Packet

Figure 2 is the program MAIN, and Figure 3 is the subroutine SUBR.

```
      DIMENSION ARRAYA(3)
      DO 25 I=1,4
      L=12
25    CONTINUE
      DC 26 I=1,3
      K=1
      CALL SUBR(ARRAYA,K)
26    CONTINUE
      STCP
      END
```

Figure 2. Program MAIN

```
      SUBRCUTINE SUBR(C,K)
      DIMENSION D(3)
      Z=6
1     DO 30 I=1,9
      D(K)=K+I+10
30    CONTINUE
      RETURN
      END
```

Figure 3. Subroutine SUBR in deck MIDDLE

Figure 4 is an example of a FORTRAN IV load-time debugging packet. The $IBDBL card heading the packet calls in the debugging compiler. Also, it specifies that all debugging activity is to cease when 450 debug requests have been executed at object time and that a maximum of 500 lines of debugging output is to be printed.

```
$IBDBL          TRAP MAX=450, LINE MAX=500
*DEBUG MAIN     25
      NAME     A/=NEW(X)
      ON 1 SET A=13
      IF L GE A RETURN
      DUMP A,L,'L TOO SMALL'
      SET A=A-1
*DEBUG MIDDLE 1
      NAME     X/=NEW(X)
      SET X=1
*DEBUG MIDDLE  30
      ON(X)    1,2,3    DUMP MAIN$$ARRAYA
*DEND
```

Figure 4. Example of a FORTRAN IV Load-Time Debugging Packet

The first debug request is executed at statement 25 in deck MAIN. The fixed-point variable A is defined for use in the debugging request for deck MAIN. The first time that statement 25 is executed, A is set to 13. L (in program MAIN) is tested each time prior to the execution of statement 25. If L is less than A, A and L are dumped, the message 'L TOO SMALL' is printed, and A is decreased by 1. If L is greater than or equal to A, return is made to the interrupted program. Then, statement 25 is executed. (.DGBTX and .IBDB4 on the dump refer to the debugging compiler output deck and the name of a control section used for definition of =NEW symbols respectively.)

The second and third debug requests are executed at statements 1 and 30 respectively, in deck MIDDLE. Suppose that deck MAIN calls subprogram MIDDLE several times and that statement 1 is the first executable statement in MIDDLE. Further suppose that MIDDLE contains a DO loop that causes statement 30 to be executed many times. Under these conditions, the second request sets the counter X to 1 each time that MIDDLE is called, and the third request dumps the array, ARRAYA, in program MAIN the third time that statement 30 in MIDDLE is executed.

The *DEND card terminates the action of the debugging compiler.

Figure 5 is the output from the debugging postprocessor that was obtained from the programs in Figures 2, 3, and 4.

## Additional Load-Time Debugging Features

The following sections contain descriptions of additional debugging facilities that are of interest mainly to the MAP programmer. They may also be used by the FORTRAN or COBOL programmer.

### *Quantities Available for Use in Debug Request Statements*

The quantities in Table I are available for use in the statements that make up the text of the debug request.

Numerical constants may be used in arithmetic expressions. A constant may be decimal floating point, decimal integer, or octal integer. If a number n contains a period, it is a floating point number; in this case, an E, EE, or D, followed by an exponent may be present. EE or D is used to indicate double precision floating point. If the first character of n (other than + or −) is a zero and n does not contain a period, an eight, or a nine, n is an octal integer and may be up to 13 digits in length. Otherwise, n is a decimal integer. Complex constants are written in the form ($n_1$, $n_2$), where $n_1$ is the real part and $n_2$ is the imaginary part. Although the machine representation is floating point, $n_1$ and $n_2$ can be octal, decimal, or floating point.

```
1,   ******** CUMP REQUEST AT 3A, REL LOC 00022, ABS LOC 04023, IN DECK MAIN

                    (A,A,X)

 .DBGTK       A          .IBDB4
 00224 05515 A                +13.

                    (L,L,X)

 MAIN         L          ......+43
 00053 04054 L                +12.
 L TOQ SMALL

2,   ******** CUMP RECUEST AT 4A, REL LOC 00056, ABS LOC 04137, IN DECK MIDDLE

                    (ARRAYA,ARRAYA+2,F)

 MAIN        ARRAYA     ......+39        (3)
 00047 04050 ARRAYA     +.31CC0000+02    -.18465318-18    -.18465318-18

3,   ******** CUMP RECUEST AT 4A, REL LOC 00056, ABS LOC 04137, IN DECK MIDDLE

                    (ARRAYA,ARRAYA+2,F)

 MAIN        ARRAYA     ......+39        (3)
 00047 04050 ARRAYA     +.91C0C000+02    +.32000000+02    -.18465318-18

4,   ******** CUMP REQUEST AT 4A, REL LCC 00056, ABS LOC 04137, IN DECK MIDDLE

                    (ARRAYA,ARRAYA+2,F)

 MAIN        ARRAYA     ......+39        (3)
 00047 04050 ARRAYA     +.91CC0000+02    +.92000000+02    +.330C0C00+02
```

●Figure 5. Example of the Output of the Debugging Postprocessor

Table I. Additional Quantities Available for Use in Statements

| QUANTITY | MEANING |
|---|---|
| =AC | Accumulator (S, 1, 2, . . . , 35) |
| =AC(i) | Accumulator bit i; $0 \leq i \leq 35$; bit 0 = S − bit |
| =AC($i_1 - i_2$) | Accumulator bits $i_1$ through $i_2$; $0 \leq i_1 \leq i_2 \leq 35$; bit 0 = S − bit |
| =LAC[5] | Logical accumulator (P, 1, 2, . . . , 35) |
| =LAC[5](i) | Logical accumulator bit i; $0 \leq i \leq 35$; bit 0 = P − bit |
| =LAC[5]($i_1 - i_2$) | Logical accumulator bits $i_1$ through $i_2$; $0 \leq i_1 \leq i_2 \leq 35$; bit 0 = P − bit |
| =MQ | Multiplier-quotient register (S, 1, 2, . . . , 35) |
| =MQ(i) | Multiplier-quotient register bit i; $0 \leq i \leq 35$; bit 0 = S − bit |
| =MQ($i_1 - i_2$) | Multiplier-quotient bit $i_1$ through $i_2$; $0 \leq i_1 \leq i_2 \leq 35$; bit 0 = S − bit |
| =SI | Sense indicator register (0-35) |
| =SI(i) | Sense indicator register bit i; $0 \leq i \leq 35$ |
| =SI($i_1 - i_2$) | Sense indicator bits $i_1$ through $i_2$; $0 \leq i_1 \leq i_2 \leq 35$; |
| =XRk | Index register k; $1 \leqslant k \leqslant 7$ |

[5]Logical accumulator may not be altered in a SET statement.

The following are examples of numerical constants:

| | |
|---|---|
| 0120 | octal integer |
| 129 | decimal integer |
| 0129 | decimal integer |
| 0.129E3 | floating point |
| (0120, −129) | complex, having true value (+80.0, −129.0) |

### Examples of the Logical IF Statement

The following examples of the logical IF clause make use of some of the quantities that refer to internal registers.

```
IF ( =SI(17) OR=SI (21))
IF ( =SI (3-7) EQ 021)
IF ( =XR4−8 EQ 3*=XR2)
```

### Redefining Symbols

The *REDEF card allows the user to change the names of symbols used in source decks to make them acceptable for use in debug requests. Included are symbols containing parentheses and MAP symbols that would be interpreted as numbers (e.g., 0.1E3, 4.6EE2, 633.D4, 1.2). The general form of the *REDEF card is:

```
1        8             16-72
*REDEF   deckname      old1bASbnew1 [,old2bASbnew2, . . .]
```

where deckname is the deck in which the symbol to be redefined appears; b represents one or more blanks; the old$_i$ are the symbols to be redefined; and the new$_i$ are the new names of the symbols.

The variable field (columns 16-72) of a *REDEF card may be extended over more than one card by using the *ETC card which is described in the section "Load-Time Debug Requests."

*REDEF cards, redefining symbols, must precede any use of the symbol new$_i$.

### General NAME Statement

The form of the general NAME statement is:

NAME symbol$_1$/ $\left\{\begin{array}{c}\text{location}\\ \text{=NEW}\end{array}\right\}$ [(mode [(dimensions)])]

$\left[\text{, symbol}_2/\left\{\begin{array}{c}\text{location}\\ \text{= NEW}\end{array}\right\}\right.$ [(mode[(dimensions)])] . . . $\Big]$

where the parameters are defined as follows:

| | |
|---|---|
| symbol | Any valid MAP symbol not containing parentheses. |
| location | A location designation as follows:<br>1. A nonsubscripted symbol (plus or minus an integer, if desired).<br>2. =Rn, a relative location where n is an unsigned octal integer.<br>3. =An, an absolute location where n is an unsigned octal integer. |
| =NEW | =NEW specifies that a location of octal format or an area corresponding to the specified mode and dimensions is to be generated for the symbol. |
| mode (dimensions) | Identical to those given in the section "Supplying Modal Information to the Debugging Dictionary." For further explanation of dimensional information, see the section entitled "Storage Allocation Pseudo-operations" in the publication *IBM 7090/7094 IBSYS Operating System, Version 13: Macro Assembly Program (MAP) Language*, Form C28-6392. |

The general form of the NAME statement may be used not only to define symbols for use within debug requests but also to allow the use of symbols within the source program where no debugging dictionary, or an insufficient one, has been supplied. In addition, this form of the NAME statement supplies alternate modal and/or dimensional information to a symbol.

### KEEP Pseudo-Operation—For MAP Assemblies

The KEEP pseudo-operation permits the programmer to specify a debugging dictionary that contains only those symbols he wishes to use in debug requests. The format of the KEEP pseudo-operation is:

| NAME FIELD | OPERATION FIELD | VARIABLE FIELD |
|---|---|---|
| Blanks | KEEP | One or more symbols, separated by commas |

The symbols in the variable field are entered into the debugging dictionary along with any modal and dimensional information that was supplied in BSS, BES, EQU, and SYN pseudo-operations. Any number of KEEP pseudo-operations may appear in a program. If the NODD option was specified on the $IBMAP card, the KEEP pseudo-operation is ignored.

Qualified symbols may not be used in the variable field. If they occur, an error message is issued and the innermost name is used (e.g., A$B$C, C is used). If the same name appears in several qualification sections, the first encountered in the deck is used.

### Trap Transfers to Subroutines Containing Debug Requests

If a trap occurs during the execution of a debug request, the routine to which control is thereby passed should not execute another debug request. Such a relationship between debug requests is called "nesting." An error message will result from the attempted execution of the second debug request and execution of the entire job will terminate.

**_Supplying Modal Information to the Debugging Dictionary_**

IBFTC automatically produces a debugging dictionary containing modal and dimensional information. However, the MAP programmer must supply this information himself in certain cases, i.e., when using BSS, BES, EQU, and SYN. (Only modal information may be given with a BES; dimensional data is ignored.) This information is specified in additional subfields of the variable field of these operations, as follows:

| NAME FIELD | OPERATION FIELD | VARIABLE FIELD |
|---|---|---|
| Symbols | BSS<br>BES<br>EQU<br>SYN | Value<br>[, mode [($d_1$, $d_2$, $d_3$)]] |

where symbol and value are the standard forms for these operations; mode is one of O, F, X, D, J, L, S, C, or H, as described in the discussion of the LIST statement; and $d_1$, $d_2$, and $d_3$ are the dimensions[6], if any, of the array denoted by the symbol. The following are examples of these statements:

```
A      BSS     25, F(5, 5)
B      EQU     A+10, F(5, 3, 2)
C      SYN     A+6, X
```

**_Address Computation_**

Address computation is a generalized form of indirect addressing. It is of special importance when complex tables or buffer chains must be manipulated. The notation for address computation is:

=C (base address, op1, op2, . . .)

Address computation is essentially a set of chaining operations. Each operation acts on the result (called the effective address) of the preceding operation. Any address (e.g., a statement number, X, SYM(I+3), =R10, =A703, but not X+3) can be used as a base address, and any machine register can be used as a base address if an extraction operation follows. The extraction operations are ($i_1-i_2$), ADDR, and DECR, where $0 \leqq i_1 \leqq i_2 \leqq 35$. The operation ($i_1-i_2$) extracts bits $i_1$ through $i_2$ of the word located at the address specified by the preceding operation; the bits thus extracted become the address used by the next operation. If $i_2-i_1+1>15$, only the rightmost 15 bits are used. ADDR and DECR are short forms of (21-35) and (3-17), respectively. Thus, =C(A,ADDR) denotes the address portion of A.

The operation COMPL complements the current effective address to form the next effective address.

The operators +, −, *, and /, have special meanings as shown in the following examples using the operator +:

| | |
|---|---|
| +A | means "add the *address* of A." |
| +A, +B | means "add the address of A, then add the address of B." |
| +A+B | means "add the *value* found by adding the contents of A to the contents of B." |
| +n | (where n is an integer) means "add the *value* n." |

Thus, =C(A,+B) is the address of A plus the address of B, while =C(A, + B+0) or =C(A, + =C(B)) is the address of A plus the contents of B. Subtraction, multiplication, and division work similarly.

The following are examples of address computation:

| | |
|---|---|
| =C(A) | the address of A |
| DUMP =C(A) | same as DUMP A |
| SET =C(A) = =C(B)+2 | same as SET A = B+2 |
| =C(A, ADDR) | the address portion of the word at address A |

The statement

DUMP =C(A, ADDR)

dumps (in octal) the word whose location is specified in the address portion of the word at location A.

Suppose that A contains

```
PZE     B, , 12
```

then the statement

```
DUMP (=C(A, ADDR),
                 =C(A, ADDR, +=C(A, DECR), -1))
```

dumps B through B+11.

The statement

DUMP =C(A, +B)

dumps the contents of the location computed by adding the address of A to the address of B. This is usually only meaningful if one of these addresses is absolute (i.e., not a relative value that is adjusted by the Loader).

The statement

DUMP =C (=A3, −=XR4)

or the statement

DUMP =C( =A77775, +=XR4, COMPL)

dumps the contents of the word at three plus the complement of index register 4.

The statement

DUMP =C(3, −=XR4)

dumps the contents of ***statement number*** 3 plus the complement of index register 4.

**_Bit Extraction_**

In arithmetic expressions (e.g., in SET, IF, CALL, and ON statements and also in address computation and in subscripts), it is convenient to be able to handle partial word operations. The notation to be used is:

=C(address computation) (bit specification)

or

=mr(bit specification)

where the bit specification is $i_1-i_2$, ADDR, or DECR, with $0 \leqq i_1 \leqq i_2 \leqq 35$. mr denotes AC, LAC[7], MQ, SI, or XRk. The notation

=mr(i)

where $0 \leqq i \leqq 35$ is also permitted.

The following are examples of partial word operations:

[6] Arrays are structured and referenced as in FORTRAN.

[7] The logical accumulator may not be altered in a SET statement.

| | |
|---|---|
| =C(A)(DECR) | the decrement portion of A |
| =C(A)(18-20) | the tag of A |
| =AC(0) | the sign of the accumulator |
| =SI(17) | bit 17 of the sense indicators |
| =C(=AC)(18-20) | invalid |

The statement

SET =C(A)(DECR) = =C(B)(18-20)+6

replaces the decrement of the word at A with the sum of 6 and the tag of B.

The statement

IF =SI(4-7) EQ 012 SET =AC(ADDR)=Q

replaces the address portion of the accumulator with the address portion of the word Q (the rest of the accumulator is not affected) if sense indicator bits 4 and 6 are 1, and bits 5 and 7 are 0.

The statement

DUMP A(=C(B)(10-17))

dumps the jth element of the array A, where j is the number found in bits 10 through 17 in the word at location B.

### A MAP Load-Time Debugging Packet

The numbers across the top of the example in Figure 6 indicate the card columns in which the various fields begin.

The $IBDBL card heading the packet specifies that all debugging activity is to cease when 1,000 requests have been executed and that a maximum of 500 lines of debugging output is to be printed.

The first request is executed the first time and each time thereafter up to but not including the eleventh time that statement number 34 in DECKA is to be executed. This request specifies that the elements in LIST statement numbers 2 and 3 (which appear later in the packet) are to be dumped.

The second request is executed prior to each execution of statement number 42 in DECKA. It tests the value of A-B*C against 3.5*D and returns to the interrupted program if they are not equal. If they are equal, it dumps the range (in complex number format) from U through V (if V is an array, the whole array will be dumped), $\text{DARRAY}_{1,7,3}$, and the message A-B*C EQ 3.5*D CAUSED PROGRAM STOP. Program execution then stops and the postprocessor is called.

The second request also contains LIST statement number 3, which, when used in a DUMP statement, causes the following information to be dumped: $\text{ALPHA}_{8,1}$, $\text{ALPHA}_{8,2}$, $\text{ALPHA}_{9,1}$, $\text{ALPHA}_{9,2}$, . . . , $\text{ALPHA}_{11,2}$; all of the elements of BARRAY; and all of blank COMMON.

The third request is executed prior to each execution of the instruction at START+17 in DECKB. On the third, sixth, ninth, twelfth, fifteenth, and eighteenth times that variable LOGVAR is true (i.e., nonzero) the following will be dumped: X; control section CNTRLA; the region from relative locations 6 through $103_8$, in octal format; and the elements in LIST statement number 4, which is contained in another debug request.

The fourth request is executed on the hundredth time that statement number 34 in DECKA is to be executed. At all other times, it simply returns control to the interrupted program. The information dumped consists of the console registers and indicators (LIST statement number 2) and the elements specified in LIST statement number 3, which is contained in the second request.

Referring to Figure 6, note that in the lines beginning ***ERROR APPROXIMATELY AT V the V is an arrow pointing to the approximate part of the source statement in which an error has occurred.

The fourth request also contains LIST statement number 4, which, when used in a DUMP statement, causes the principle diagonal of the matrix CARRAY (in DECKA) to be dumped.

The fifth request is executed the first time and every second time thereafter through the ninth time that the instruction at RESTRT in DECKB is to be executed. It tests variable Q and, if Q is negative, dumps the elements in LIST statement number 2 (which is contained in the fourth request); the message Q LESS THAN 0; and the elements in LIST statement number 6 which specifies the region from RESTRT through RESTRT + 100 in symbolic format.

The sixth request is executed prior to each execution of statement numbers 37 and 42 in DECKA. If A is less than B, it simply returns control to the interrupted program; otherwise it dumps A, B, $E_1$, $E_2$, and $E_3$.

```
    1    678       16


    $IBDBL         TRAP MAX=1000, LINE=500
    *DEBUG DECKA   34
          ON 1,10 DUMP 2,3
    *DEBUG DECKA   42
          IF A-B*C NE 3.5*D RETURN
          DUMP (U,V,J),DARRAY(1,7,3),
         X 'A-B*C EQ 3.5*D CAUSED PROGRAM STOP'
          STOP
    3     LIST ((ALPHA(I,J),J=1,2),I=8,11),
         X  BARRAY,//
    *DEBUG DECKB   START+17
          IF (LOGVAR) ON 3,18,3 DUMP X,/CNTRLA/,
         X (=R6,=R103,0),4
    *DEBUG DECKA   34
          ON 100 GO TO 10
          RETURN
    10    DUMP 2,3
          DUMP       'DATA',=AC,=XR1,=R1(S),=A117(0)
*** ILLEGAL LIST ELEMENT. ELEMENT DELETED.
*** ERROR APPROXIMATELY AT    V
                     AC,=XR1,=R1(S),=A117(0)
*** ILLEGAL LIST ELEMENT. ELEMENT DELETED.
*** ERROR APPROXIMATELY AT    V
                     R1(S),=A117(0)
          STOP
    2     LIST CONSOLE
    4     LIST (CARRAY(I,I),I=1,10)
    *DEBUG DECKB   RESTRT
          ON 1,10,2 IF Q LT 0.0 DUMP 2,
         X 'Q LESS THAN 0',6
    6     LIST (RESTRT,RESTRT+100,S)
    *DEBUG DECKA   37,42
          IF (A .LT. B) RETURN
          DUMP A,B,(E(J),J=1,3)
    *DEND
```

Figure 6. Example of a MAP Load-Time Debugging Packet

## Index

IBM

**International Business Machines Corporation**
**Data Processing Division**
**1133 Westchester Avenue, White Plains, New York 10604**
**[U.S.A. only]**

**IBM World Trade Corporation**
**821 United Nations Plaza, New York, New York 10017**
**[International]**

Printed in U.S.A. GC28-6393-2